

# **A Framework For Instrument Monitoring On The Grid**

**Stuart Kenny**

A thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

September 2005

## Declaration

This thesis has not been submitted as an exercise for a degree at any other University. Except where otherwise stated, the work described herein has been carried out by the author alone. This thesis may be borrowed or copied upon request with the permission of the Librarian, University of Dublin, Trinity College. The copyright belongs jointly to the University of Dublin and Stuart Kenny.

---

Stuart Kenny

Dated: September 19, 2005

## Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

---

Stuart Kenny

Dated: September 19, 2005

# Acknowledgements

Firstly I must thank my parents for supporting me, in every way, throughout all my years at college. I would also like to thank my supervisor Brian Coghlan, and all those in the Computer Architecture Group, for their help, input and guidance. This research could not have taken place without the assistance of all the R-GMA developers, who kindly allowed me to work with them, and provided significant help when needed. Much of this research was funded by CrossGrid, and I would like to thank all the CrossGrid members who helped me during the three years of that project.

***Stuart Kenny***

*University of Dublin, Trinity College*

*September 2005*

# Contents

<b>Acknowledgements</b>	<b>iv</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>vii</b>
<b>List of Abbreviations</b>	<b>viii</b>
<b>Summary</b>	<b>xi</b>
<b>Chapter 1 INTRODUCTION</b>	<b>1</b>
1.1 OVERVIEW . . . . .	1
1.2 OBJECTIVES . . . . .	3
1.3 THESIS STRUCTURE . . . . .	4
<b>Chapter 2 STATE OF THE ART</b>	<b>6</b>
2.1 DISTRIBUTED MONITORING TOOLS . . . . .	7
2.1.1 Network Weather Service . . . . .	7
2.1.2 Ganglia . . . . .	8
2.1.3 NetLogger . . . . .	9
2.2 GRID INFORMATION SYSTEMS . . . . .	11

2.2.1	Globus Monitoring and Directory Service . . . . .	11
2.2.2	GridICE . . . . .	13
2.2.3	Legion . . . . .	14
<b>Chapter 3</b>	<b>R-GMA</b>	<b>16</b>
3.1	THE GRID MONITORING ARCHITECTURE . . . . .	16
3.2	THE R-GMA . . . . .	17
3.2.1	Query types and Producer types . . . . .	18
3.2.2	Architecture . . . . .	20
<b>Chapter 4</b>	<b>THE DATAGRID AND CROSSGRID PROJECTS</b>	<b>23</b>
4.1	DATAGRID . . . . .	23
4.2	CROSSGRID . . . . .	24
4.3	PROJECT INTERACTIONS . . . . .	26
<b>Chapter 5</b>	<b>THE SANTA-G MONITORING FRAMEWORK</b>	<b>27</b>
5.1	FRAMEWORK PURPOSE . . . . .	27
5.2	THE CANONICAL PRODUCER . . . . .	28
5.3	FRAMEWORK ARCHITECTURE . . . . .	31
<b>Chapter 6</b>	<b>DESIGN AND ARCHITECTURE</b>	<b>34</b>
<b>Chapter 7</b>	<b>IMPLEMENTATION</b>	<b>42</b>
7.1	THE PUBLISHING MODULE . . . . .	42
7.1.1	The Sensor . . . . .	43
7.1.2	The QueryEngine . . . . .	50
7.1.3	The Viewer module . . . . .	60

<b>Chapter 8 TESTING</b>	<b>67</b>
8.1 TEST DEPLOYMENT . . . . .	67
8.2 FUNCTIONAL TESTS . . . . .	69
8.3 PERFORMANCE TESTS . . . . .	71
8.4 RESULTS . . . . .	71
8.4.1 Time for ResultSet retrieval . . . . .	71
8.4.2 Time for packet searching . . . . .	75
8.4.3 Time per component . . . . .	78
8.5 RESULTS DISCUSSION . . . . .	80
<b>Chapter 9 EXAMPLE EXPERIMENTS</b>	<b>84</b>
9.1 TCP THROUGHPUT MEASUREMENTS . . . . .	84
9.1.1 Configure the SANTA-G system . . . . .	85
9.1.2 Write the Consumer code . . . . .	87
9.1.3 Run the experiment . . . . .	92
9.2 MPI RING MEASUREMENTS . . . . .	96
9.2.1 Configure the SANTA-G system . . . . .	96
9.2.2 Write the Consumer code . . . . .	97
9.2.3 Run the experiment . . . . .	98
9.3 ONE-WAY LATENCY MEASUREMENTS . . . . .	105
9.3.1 Configure the SANTA-G system . . . . .	105
9.3.2 Write the Consumer code . . . . .	105
9.3.3 Run the experiment . . . . .	106
9.4 SCI TRACE ANALYSIS . . . . .	120
9.4.1 Configure the SANTA-G system . . . . .	120
9.4.2 Write the Consumer code . . . . .	121
9.4.3 Run the experiment . . . . .	121

<b>Chapter 10 GRID-WIDE INTRUSION DETECTION</b>	<b>132</b>
10.1 INTRUSION DETECTION . . . . .	133
10.2 EXISTING APPROACHES . . . . .	134
10.3 GRID-WIDE INTRUSION DETECTION . . . . .	136
10.3.1 Snort . . . . .	136
10.3.2 NetTracer Snort sensor . . . . .	137
10.3.3 GIDS Design . . . . .	139
10.4 DISCUSSION . . . . .	143
<b>Chapter 11 INTRUSION DETECTION EXAMPLE</b>	<b>146</b>
11.1 TEST DEPLOYMENT . . . . .	147
11.2 EXAMPLE ANALYSERS . . . . .	148
11.2.1 Simple Pattern Matching Analyser . . . . .	149
11.2.2 Heuristic-based Analyser . . . . .	150
11.2.3 Anomaly-based Analyser . . . . .	151
11.3 EXAMPLE INTRUSION DETECTION . . . . .	152
11.4 GRID-IRELAND DEPLOYMENT . . . . .	157
<b>Chapter 12 FUTURE WORK</b>	<b>165</b>
12.1 SENSORS AND QUERYENGINES (INSTRUMENTS) . . . . .	165
12.2 POSTPROCESSORS (ANALYSERS) . . . . .	169
<b>Chapter 13 CONCLUSION</b>	<b>173</b>
13.1 OVERVIEW . . . . .	173
13.2 PERSONAL SUMMARY . . . . .	173
<b>Bibliography</b>	<b>175</b>

# List of Figures

3.1	Grid Monitoring Architecture . . . . .	16
3.2	A possible topology of R-GMA components . . . . .	20
3.3	Relational Grid Monitoring Architecture . . . . .	22
4.1	The grid monitoring system . . . . .	25
5.1	CanonicalProducer servlet communication . . . . .	29
5.2	Monitoring framework . . . . .	31
5.3	An example of CanonicalProducer user code . . . . .	32
6.1	NetTracer publishing module . . . . .	36
6.2	Example NetTracer deployment . . . . .	37
7.1	Publishing module implementation structure . . . . .	42
7.2	Tcpdump Sensor startup sequence . . . . .	44
7.3	Sensor class diagram . . . . .	48
7.4	Remote file server . . . . .	49
7.5	QueryEngine startup sequence . . . . .	51
7.6	QueryEngine query processing sequence . . . . .	54
7.7	QueryEngine class diagram . . . . .	55

7.8	EthernetFilter class diagram . . . . .	57
7.9	SCIFilter class diagram . . . . .	58
7.10	Viewer query submission sequence . . . . .	61
7.11	Ethernet packet display process . . . . .	62
7.12	Viewer GUI, packet view panel . . . . .	63
7.13	Viewer GUI, query view panel . . . . .	64
7.14	Viewer GUI, sensor information panels . . . . .	65
7.15	Viewer GUI, query builder . . . . .	66
7.16	Viewer GUI, Snort alerts panel . . . . .	66
8.1	Test deployment . . . . .	68
8.2	Average times for retrieval . . . . .	80
8.3	Average times for retrieval of tuples from file positions . . . . .	81
8.4	Time spent in each component during query execution . . . . .	82
9.1	Throughput measurements deployment . . . . .	86
9.2	TCP throughput Consumer components . . . . .	89
9.3	Throughput Consumer sequence diagram . . . . .	90
9.4	Direction of MPI ring . . . . .	97
9.5	Captured MPI packets in Viewer GUI . . . . .	100
9.6	MPI Ring and NetTracer effective bandwidth distributions . . . . .	102
9.7	MPI Ring and NetTracer latency distributions . . . . .	103
9.8	Overhead per site . . . . .	104
9.9	$T_{owl}$ measurements for single application execution . . . . .	108
9.10	$T_{offset}$ measurements for single application execution . . . . .	109
9.11	Mean $T_{owl}$ values, series 1 . . . . .	110
9.12	$T_{owl}$ minimum, maximum and standard deviation, series 1 . . . . .	111
9.13	Mean $T_{owl}$ values, series 1 and 2 . . . . .	112

9.14	$T_{owl}$ minimum, maximum, and standard deviation, series 2 . . . . .	113
9.15	$T_{owl}$ measurements, series 3 . . . . .	114
9.16	Mean $T_{owl}$ values over 72 hour measurement period . . . . .	115
9.17	$T_{offset}$ measurements, series 1 . . . . .	117
9.18	$T_{offset}$ measurements, series 2 . . . . .	118
9.19	Mean $T_{offset}$ values, series 3 . . . . .	119
9.20	Mean $T_{offset}$ values over 72 hour measurement period . . . . .	119
9.21	SCI trace analysis deployment . . . . .	121
9.22	SCI trace analysis tool . . . . .	122
9.23	SCI transaction . . . . .	123
9.24	Packet inter-arrival time PDF . . . . .	124
9.25	Packet size PDF . . . . .	125
9.26	Throughput in bytes/sec . . . . .	126
9.27	Throughput in bytes/sec, start and end of trace . . . . .	127
9.28	Packet inter-arrival time series . . . . .	128
9.29	Packet inter-arrival time PDF, 2-d torus . . . . .	129
9.30	Distribution of accesses versus source and target IDs . . . . .	131
10.1	Snort architecture . . . . .	137
10.2	NetTracer Snort monitoring . . . . .	138
10.3	Multiple sites stream alerts to the R-GMA . . . . .	140
10.4	Archiver collects alerts to grid-wide intrusion log . . . . .	140
10.5	Intrusion log analysis by alert Consumers . . . . .	142
11.1	GIDS test deployment . . . . .	147
11.2	Distribution of alerts by site, excluding the SNMP alerts at NUIM . . . . .	158
11.3	SSH alert viewed using Viewer GUI . . . . .	163

12.1 Secure NetTracer prototype structure . . . . .	167
---	-----

# List of Tables

8.1	Time for retrieval of 1 tuple . . . . .	72
8.2	Time for retrieval of 10 tuples . . . . .	73
8.3	Time for retrieval of 100 tuples . . . . .	73
8.4	Time for retrieval of 1000 tuples . . . . .	73
8.5	Time for retrieval of 10000 tuples . . . . .	74
8.6	Average times for retrieval . . . . .	74
8.7	Time for retrieval of tuples from beginning of file . . . . .	75
8.8	Time for retrieval of tuples from middle of file . . . . .	76
8.9	Time for retrieval of tuples from end of file . . . . .	76
8.10	Average times for retrieval of tuples from file positions . . . . .	77
8.11	Times for retrieval of 1 tuple . . . . .	78
8.12	Times for retrieval of 10 tuples . . . . .	78
8.13	Times for retrieval of 100 tuples . . . . .	79
8.14	Times for retrieval of 1000 tuples . . . . .	79
8.15	Times for retrieval of 10000 tuples . . . . .	79
9.1	Sample throughput values obtained . . . . .	90
9.2	Throughput values obtained during query submission . . . . .	93
9.3	Connections and the API calls associated with them . . . . .	94

9.4	SQL query transmission measurement . . . . .	94
9.5	Summary of calculated times (ms) . . . . .	95
9.6	Data collected by sensors for single ring transmission . . . . .	101
9.7	Average overhead per site . . . . .	104
9.8	Collected dataset series . . . . .	110
9.9	Distribution of target address accesses . . . . .	130
10.1	Snort alerts table schema . . . . .	139
11.1	Sample of TCP traffic gathered by sensor during attack on TestGrid . . . . .	153
11.2	Positive responses to port scan on TestGrid . . . . .	153
11.3	Sample of alerts logged to TestGrid grid-wide intrusion log . . . . .	154
11.4	Distribution of alerts by Grid-Ireland site . . . . .	157
11.5	MS SQL alert pattern . . . . .	159
11.6	Distribution of alert types . . . . .	161

# List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
AI	Artificial Intelligence
AIDE	Advanced Intrusion Detection Environment
API	Application Programming Interface
CE	Computing Element
CERN	European Centre for Nuclear Research
CSR	Control and Status Register
DIDS	Distributed Intrusion Detection System
DoS	Denial of Service
DDoS	Distributed Denial of Service
EDG	European DataGrid
EGEE	Enabling Grids for e-Science
FSA	Fingerprint Sharing Alliance
GGF	Global Grid Forum
GIIS	Grid Index Information Service
GLUE	Grid Laboratory Universal Environment
GMA	Grid Monitoring Architecture
GOC	Grid Operations Centre
GPS	Global Positioning System
GRIS	Grid Resource Information Service
GIS	Grid Security Infrastructure

GUI	Graphical User Interface
HIDS	Host-based Intrusion Detection System
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISP	Internet Service Provider
JIMS	JMX-based Information Monitoring System
JMX	Java Management Extensions
LCG	LHC Computing Grid Project
LCFGng	Local ConFiGuration System next generation
LDAP	Lightweight Directory Access Protocol
LHC	Large Hadron Collider
MAC	Media Access Control
MDS	Monitoring and Directory Service
MPI	Message Passing Interface
MPICH	Portable implementation of MPI
MPICH-G2	Grid-enabled MPI implementation
NFS	Network File System
NIDS	Network-based Intrusion Detection System
NREN	National Research and Education Network
NWS	Network Weather Service
OCM	OMIS Compliant Monitoring system
OCM-G	Grid-enabled OMIS Compliant Monitoring system
OS	Operating System
PBS	Portable Batch System
PCI	Peripheral Component Interconnect
PDF	Probability Density Function
PHP	Hypertext Preprocessor
QoS	Quality of Service

RDBMS	Relational Database Management System
R-GMA	Relational Grid Monitoring Architecture
RIPE NCC	Reseaux IP European Network Coordination Center
RSL	Resource Specification Language
RTT	Round Trip Time
SAN	System Area Network
SANTA	System Area Networks Trace Analysis
SANTA-G	Grid-enabled System Area Networks Trace Analysis
SCI	Scalable Coherent Interface
SE	Storage Element
SQL	Structured Query Language
TCP	Transmission Control Protocol
TTM	Test Traffic Measurements
UDP	User Datagram Protocol
UI	User Interface
ULM	Universal Logger Format
UML	Unified Modeling Language
VO	Virtual Organisation
WN	Worker Node
XDR	External Data Representation
XML	Extensible Markup Language

# Summary

Grid computing allows sharing of geographically distributed resources. It enables the selection and aggregation of a wide variety of geographically distributed resources as a single unified computing resource for solving large scale compute and data intensive computing applications.

As with any computer system an important task within a grid is monitoring. The ability to monitor distributed resources is crucial to high performance computation. Amongst other things, it allows one to evaluate behaviour, optimize behaviour, discover and diagnose problems or faults.

The objective of this research was to design a framework that would provide a generic template to allow for ad-hoc monitoring experiments with external instruments in a grid environment. The template allows for the information captured by external instruments, either hardware or software, to be accessed through a grid information system. Monitoring instruments in general create a huge amount of monitoring data that is often stored in raw log files. The sheer size of the data generated makes it unsuitable for direct insertion into an information system. The idea of the framework is to make this data accessible through an information system whilst allowing the data to remain in-situ.

A demonstrator of the framework was also to be implemented as part of this research. To accomplish this, first the central component of the framework, the interface to the grid information system, in this case R-GMA, which is a relational implementation of the Global Grid Forum's Grid Monitoring Architecture developed within the EU DataGrid project, had

to be designed and implemented. This resulted in the development of a new type of R-GMA producer, the Canonical Producer.

The proposed framework demonstrator was a network tracer (NetTracer) that would allow access to monitoring data obtained from a set of example network monitoring instruments through R-GMA. The instruments chosen support the tracing of two network interconnect technologies, Ethernet through Tcpdump, a software network packet capture application, and SCI (Scalable Coherent Interface), using a (hardware) SCI trace instrument. The third (software) network monitoring tool supported is Snort, a network-based intrusion detection system.

The research was successful in its objectives. The framework was designed and the NetTracer demonstrator implemented shows that is a viable concept. The research has contributed to three major grid projects. The Canonical Producer, the enabling technology for the framework, is now part of the R-GMA system. The initial implementation of the NetTracer demonstrator was developed within the EU Crossgrid project, and forms part of its grid monitoring system. The NetTracer is also being used by Grid-Ireland, the national computational grid of Ireland, in order to monitor network activity on its sites and also as the basis of a grid-wide intrusion detection system.

# Chapter 1

## INTRODUCTION

### 1.1 OVERVIEW

In order to cope with the increasing amount of processing power required by modern applications, processors have been becoming ever more powerful and fast. Eventually, however, single processor systems, and even multiprocessor systems, will reach a limit defined by such factors as cost and physics. Although supercomputers can be used to achieve improved performance, they are extremely expensive and this makes their use prohibitive, especially in research. A solution to this is cluster systems, i.e. nodes interconnected by System Area Networks (SAN), which provide close to supercomputer performance but at a fraction of the cost. Clusters use many ‘off the shelf’ PC’s connected by a SAN to provide ‘a single unified computing resource’ [31].

Although the use of cluster systems is common in universities, in many cases the average computing environment still remains inadequate for large scale compute and data intensive applications [13]. Computational grids are intended to provide a solution to this. [13] defines computational grids as ‘a hardware or software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities’. They do

this by sharing geographically distributed resources such as single computers, clusters, super-computers, data or instruments. Several projects are currently developing grid environments. The DataGrid project, a very influential project that completed in March 2004, was created to tackle the problem of how to store, move and analyse the huge amounts of data created by the latest high energy physics experiments, such as the Large Hadron Collider (LHC) at CERN. The aim of the CrossGrid project, which ended in April 2005, was to create tools and services that support interactive applications in the grid environment, i.e. applications with which users will be able to interact and obtain results in real time.

An essential task within the grid, as in any large scale distributed system, is monitoring. The ability to monitor distributed resources is crucial to high performance computation. It allows one to:

- find the cause of performance problems
- tune systems parameters in order to optimise resource usage
- detect and diagnose problems or faults
- optimise job scheduling
- perform billing and accounting

Moreover, if a timestamp is added to any item of information, a grid monitoring framework can become a general purpose grid information system.

A grid information system is used to provide information on the current state of the grid's resources. Some examples of grid information systems currently in use are Globus MDS, GridICE, the Legion Resource Directory Service, and the DataGrid R-GMA. The Relational Grid Monitoring Architecture (R-GMA) is the information system used for this research. It is based on the Global Grid Forum's (GGF) Grid Monitoring Architecture (GMA), a general architecture for grid monitoring systems.

## 1.2 OBJECTIVES

The motivation for this research came from the Computer Architecture Group's (CAG) (located at Trinity College Dublin) efforts to 'systematize the collection and analysis of interconnect traces via relational database technologies', referred to as SANTA (System Area Networks Trace Analysis). The work carried out by CAG, within the SCIEurope project, resulted in the development of an SCI (Scalable Coherent Interface) tracer/analyser and a set of software tools for the acquisition and analysis of deep non-invasive SCI interconnect traces. The aim of this research is to extend this concept to the grid.

The difficulty with this approach is that, in general, the typical grid information system model of information providers periodically inserting data is not suitable for this type of instrument monitoring. When dealing with information sources that produce large amounts of data, the SCI tracer for example, it may be inefficient, or impossible, to insert all of the data into the information system. It would be better to leave the data where it was created, and to only transfer selected subsets of the data when specifically requested by a user. The objective of this research was to design a framework to allow for this.

The framework, known as SANTA-G (grid-enabled SANTA), provides a generic template to allow for ad-hoc monitoring experiments with external instruments in the grid environment. The idea of the framework is to make the monitoring data accessible through the grid information system whilst allowing the data to remain in-situ.

After the framework was defined, it was proposed that a demonstrator should be designed and implemented. This demonstrator, developed within the EU CrossGrid project, is known as the NetTracer. The NetTracer demonstrates SANTA-G by providing access to log files created by network monitoring instruments through the R-GMA information system. R-GMA was chosen, amongst other reasons, for its use of the relational model, and hence its compatibility with the original SANTA concept. The enabling technology for the framework (developed as part of this research as a component of R-GMA) is known as the Canonical

Producer [7]. The example instruments supported by the NetTracer are Tcpdump, a software instrument for ethernet network tracing, Snort, a software-based network intrusion detection system, and the SCI tracer, a hardware instrument, as described above.

The acquisition of the monitoring data represents only the first stage in the SANTA-G framework. The use of R-GMA allows for the analysis of the raw data gathered by the NetTracer through the use of custom R-GMA Consumers, which can be used to obtain subsets of the available data with SQL SELECT statements, as if querying a relational database. As part of this research several example analysers of NetTracer logs were developed, for example, analysers that calculate throughput and one-way latency times from Tcpdump logs of TCP/IP traffic. Analysers of Snort logs were used to create a grid-wide intrusion detection system, the initial design and implementation of which was also carried out as part of this research.

### **1.3 THESIS STRUCTURE**

The thesis first provides some background to the research. It then goes on to describe the framework, and also the design and implementation of the framework demonstrator, the NetTracer. The results of testing of the NetTracer are then given along with some example experiments that utilise the NetTracer. The initial design and implementation of the grid-wide intrusion detection system is then discussed, along with an example of its use in a grid environment. This is followed by some suggestions for further work, and also of the outcome of the research.

Chapter 2 provides a state of the art review of both monitoring tools and information systems. A description of R-GMA is given in Chapter 3. The CrossGrid project, within which the NetTracer was developed, is described in Chapter 4. Chapter 5 introduces the framework that is the subject of the thesis. The design and implementation of the NetTracer is presented in Chapters 6 and 7. A description of NetTracer testing along with the results

of these tests is provided in Chapter 8. Some example NetTracer experiments are described in Chapter 9. Chapters 10 and 11 contain a description of the initial work carried out on the design and implementation of the intrusion detection system, along with some initial testing and results. A discussion of possible future work is given in Chapter 12.

## Chapter 2

# STATE OF THE ART

This chapter describes a macroscopic state of the art for two separate areas: distributed monitoring tools and grid information systems. The first section deals with distributed monitoring tools, with a particular emphasis on those used for network and resource monitoring. The tools described are the Network Weather Service, Ganglia and NetLogger.

The second section describes some of the current systems used as grid information systems. The Globus Monitoring and Directory Service, the GridICE Information System, and the Legion Resource Directory Service are described. A fourth important system, R-GMA, is described in the next chapter.

A more detailed state of the art is not appropriate, as this thesis describes the creation of a framework for ad-hoc non-invasive monitoring with external instruments in a grid environment, not the monitoring or grid information system nor the instruments. No such framework existed, and as yet the framework is unique. Here I describe the important background, not the prior art.

## 2.1 DISTRIBUTED MONITORING TOOLS

### 2.1.1 Network Weather Service

In a distributed system being utilised by a large number of users it is important that users have the ability to choose resources that are the most lightly loaded when submitting their applications. When making this decision it is not the current load that should be used, but the estimated load in the near future, i.e. when they will submit their application to be executed. The Network Weather Service (NWS) [44] provides a system that can be used to generate, or forecast, short-term performance measurements of network and computing resources in large scale distributed systems. In order to deal with the difficulties of monitoring and performance measurement in these types of systems, the NWS uses adaptive programming techniques, distributed fault-tolerant control algorithms, and an extensible system architecture.

The NWS is composed of four component processes:

**persistent storage process:** used to store and retrieve data from persistent storage. Data is stored persistently as a text string, which can be associated with a timestamp. Because the NWS is used to provide short-term forecasts, data is not stored permanently. Instead the data is managed as a circular queue of files. If the queue fills older data will be lost. Any data to be stored indefinitely must be retrieved and stored externally.

**name server:** the name server provides a directory service, used to map human readable process and data name text strings to low-level addresses in the form of a TCP/IP address and port number. An NWS process registers its address with the Name Service, contacted by way of a well known address, the only one used by the system. This is done periodically so that processes that disappear can be removed from the directory.

**sensor:** the sensor process runs on the resource to be monitored, gathering the required performance measurements. A timestamp is appended to every measurement taken.

Examples of the sensors provided by the NWS are the CPU Sensor and the Network Sensor. The Network Sensor uses active network probes to obtain measurements such as small-message round-trip time, large message throughput, and TCP socket connect-disconnect time.

**forecaster:** in order to generate a forecast, the forecaster process obtains the relevant data from a persistent state process. This will be the most recent data available due to the way in which the persistent state process manages the data, i.e. as a circular queue. Also, because each measurement is associated with a timestamp, it may be treated as a time series. Instead of applying a single forecasting model, a set of models is used to generate a prediction given the values contained in the series, and an error measure is calculated for that model. The model with the lowest error can then be dynamically chosen as the model to use.

If one wished to classify the usage of NWS in the community, one could say that NWS is used where prediction is required.

### 2.1.2 Ganglia

Ganglia [26] was originally developed for monitoring clusters. Its scalable design has allowed it to evolve into a distributed monitoring system that is in widespread use. Distribution is provided by using a multicast-based listen/announce protocol. In order to aggregate the state of multiple clusters Ganglia uses a hierarchical design. Data from multiple clusters is brought together by using a tree of point-to-point connections. Ganglia provides built-in metrics for node state and also allows for user-defined metrics. Examples of built-in metrics are: number of CPUs, CPU clock speed, load (1, 5, and 15 minute averages), total and free memory, total and running number of processes, total and free swap space, and operating system information such as name, version and architecture. User-defined metrics can be used to provide arbitrary application-specific state.

Ganglia is composed of two main components:

**gmond:** the Ganglia monitoring daemon is used to monitor nodes within a single cluster.

The daemon runs on each node of the cluster and provides monitoring data to clients by publishing on a well-known multicast channel. The daemon also listens on this channel for data broadcast from other nodes. Data is stored in memory in a hierarchical hash table. The daemon accepts client requests and responds to them by publishing the metrics requested in a multicast XDR format.

**gmetad:** in order to collect metrics from multiple clusters Ganglia provides the Ganglia Meta Daemon. This daemon periodically polls a set of data sources, specified in a configuration file, to obtain data. It then publishes this aggregated data in XML format to clients. The data sources can be gmond daemons, to aggregate data from nodes in a single cluster, or other gmetad daemons, in order to aggregate data from multiple clusters. The gmetad daemon can also store data for historical analysis using a Round Robin Database, managed by the RRDtool. This tool stores data to constant size databases, and also generates graphs of metrics versus time, which are published using a PHP web-front end.

Ganglia also provides a command line program, gmetric, that can be used to publish application specific metrics, and a client side library. Generally Ganglia is used to present views of past and present metrics via web-pages.

### 2.1.3 NetLogger

The Networked Application Logger [35], NetLogger, provides a methodology for network, host, and application-level monitoring, in order to allow for real-time diagnosis of performance problems. The interactions between components in a high performance system, i.e. applications, operating systems, network components, network adapters, can be very com-

plex, and can therefore make determining the cause of performance problems, such as low throughput, or high latency, very difficult. NetLogger provides tools for end-to-end instrumentation of all of these components, to allow for an overall view of the system during operation.

The NetLogger toolkit provides a number of components to do this:

**Common Log Format:** NetLogger produces timestamped logs of events, in either an ASCII or binary message format, that occur in the system. The IETF (Internet Engineering Task Force) ULM (Universal Logger Format) is used for the logging of the messages, and also for the exchange of messages. These messages are composed of a list of key-value pairs, separated by a whitespace. The binary format allows for much faster logging than the ASCII format.

**NetLogger API:** to produce logs when an interesting event occurs in an application, the NetLogger API is used to link the application to the NetLogger library. Calls to the API, such as NetLogger write(), are placed at critical points in the application code. The API provides automatic timestamping and logging of events, to either memory, a local file, or a remote host.

**netlogd:** netlogd is a daemon that allows for the collection of event logs from distributed applications. The daemon receives the log entries from the application and logs them to a single host and port.

**Monitoring tools:** as well as application monitoring using the API, NetLogger can also be used for host and network monitoring by using standard Unix system and network monitoring tools, such as netstat, vmstat, iostat, and snmpget. A NetLogger wrapper converts the output of these tools to NetLogger formatted event messages.

**Visualisation:** the NetLogger visualisation tool can be used to analyse the event logs generated by NetLogger. Three types of graph primitives are used by the visualisation

tool:

**lifeline:** represents the ‘life’ of an object as it travels through a distributed system.

The slope of the line indicates the latencies in the system.

**loadline:** a continuous segmented curve of scaled values that represents the changes in system resources such as CPU load.

**point:** shows single occurrences of events. These would usually be error or warning messages.

NetLogger is most used for instrumentation of a distributed application in order to understand its behaviour.

## 2.2 GRID INFORMATION SYSTEMS

### 2.2.1 Globus Monitoring and Directory Service

In large scale distributed systems it is necessary to have access to accurate and up-to-date information on the state of available resources, in order to allow for both selection and configuration [11]. The Globus Monitoring and Directory Service (MDS) [11] provides a uniform interface to this information, which can be collected by diverse information sources. The information sources can be any data collection service, such as Ganglia, producing either static or dynamic data. Developers can also use the MDS to provide new information providers. The MDS provides a consistent interface to both applications accessing data and the information services providing the data.

The MDS has three main components, organised in a hierarchical structure:

**Information Providers (IPs):** these form the lowest level of the hierarchy, running on the resource about which information is to be published. The IP can be used to collect data from any local data collection service. MDS provides a core set of information providers

that publish information such as CPU configuration, CPU load, operating system type, and file system and memory information. In MDS, resources, such as organisations, people, networks, or computers, are represented by objects. Instances of these objects then form entries in the MDS in order to represent a specific resource. These entries store information about the resource as attributes.

**Grid Resource Information Service (GRIS):** the GRIS also runs on the resource, and provides the gateway to the information provided by the IPs. It forms the second level in the MDS hierarchy. Clients can access aggregated data from the IPs by using the GRIS interface. For example in a cluster system being monitored using the Ganglia tool an IP can be run on each node, collecting data from the ganglia monitoring daemon. Each IP then registers with a single GRIS for the resource (i.e. compute node). Aggregated information for that node can then be obtained by querying the GRIS.

**Grid Index Information Service (GIIS):** the GIISes form the higher layers in the hierarchy, forming an aggregate directory of lower-level data [11]. Each GRIS registers with a higher level GIIS. The GIIS then requests information from the GRIS. Clients can then obtain aggregate information by querying the GIIS. It is also possible for a GIIS to register with a further higher level GIIS. For example, if a site had several clusters a site-level GIIS could be used to aggregate information from these. Another project-level GIIS could then be used to aggregate information from multiple site-level GIISes.

Originally MDS used a push model, but quickly changed to the present pull model. MDS uses the API and data representations defined by the Lightweight Directory Access Protocol (LDAP) directory service. Because of this information produced by the MDS can be viewed by using any LDAP browser. A set of PHP scripts have also been provided to allow web-based browsing of data. However MDS is constrained to the LDAP hierarchical

representations of information (schema), although multiple parallel hierarchies representing the same information are allowed. Also a hierarchy is statically defined, not dynamic.

### 2.2.2 GridICE

GridICE [3] is a monitoring infrastructure developed for use with a Grid Operations Centre (GOC). A GOC is a term defined in the LCG and EGEE grid projects for an organisation that can monitor and control a multi-institutional grid. GridICE was designed to be easily integrated with existing grid middleware.

The GridICE architecture consists of five layers:

**Measurement Service:** The measurement service forms the base layer of the architecture, used to collect data from the resources. The set of metrics defined are represented by the well-established GLUE schema, which has also been extended to include metrics related to a computer system. In the GridICE architecture a computer system is considered to have a role in a grid, e.g. a broker, that is defined by a set of processes that provide the functionality of that role. For a particular role, therefore, metrics can be obtained from this set of processes.

**Publisher Service:** The purpose of this layer is to aggregate the data obtained by the measurement service, and to provide access to the data for consumers. This is done by using MDS. For example, in a cluster, data would be collected to a single edge node with internet access, and then published to consumers through MDS.

**Data Collector Service:** A problem with MDS is that it does not provide persistent storage of monitored data. It only holds the latest value for a given metric. This does not allow for historical analysis of the data. The data collector service layer provides this. The service periodically scans MDS to determine if a new source of information has been registered and determines the metrics available from the registered sources. Values are

obtained for these metrics and persistently stored.

**Detection and Notification and Data Analyzer Services:** This is the fourth layer of the architecture and allows for users to be notified when specific events occur. Performance analysis, statistics and reports can be obtained from the Data Analyzer service.

**Presentation Service:** The final layer provides a graphical interface to users to allow them to visualise the available information. The interface is web-based and provides three different types of view:

**GOC view:** information from the entire set of resources being monitored by the GOC

**Site view:** information from the resources of a particular site

**Virtual Organisation (VO) view:** information from the set of resources to which the members of a particular VO have access.

GridICE is essentially a very extended MDS.

### 2.2.3 Legion

Legion [16] was designed to provide a complete metacomputing environment. It is based on a very nice object-oriented design that has evolved since circa 1994. The functionality of the system is provided by a set of core objects. The information system used in Legion is contained in what is referred to in the Legion architecture as service objects, a set of objects that lie between the core objects and users. The information system is a component of the resource management infrastructure. The information system has two main components, the basic resources and the information database.

**Resources:** Resources in Legion are considered to be of two types, hosts and vaults. Both are represented by core objects. The host object is used to encapsulate machine information [16] (such as processor and memory information), whereas the vault objects are

used to represent storage. All objects in Legion have persistent state associated with them, a vault object is used to store this. The host objects also contain a number of management functions, both for object and resource management, such as scheduling. Information is stored by all Legion objects as attributes. These form what is referred to as an attribute database. The host object refreshes these attributes periodically in order to accurately reflect the current state of the resource.

**Collections:** The collection performs much the same function in the Legion infrastructure as the GRIS in the Globus MDS. It collects information from the host objects, by either the push or pull model. The collection queries the host to collect information on its current state, this is the pull model. Hosts, however, can also push data to the collections of which they are aware. The collection stores the data as a set of Legion object attributes. Users can then access data on resources by querying the collection. Legion defines its own collection query language for this. The query language allows for field matching, semantic comparisons, and boolean combinations of terms [16]. All collections are capable of both sending and receiving data from other collections. This allows for the combination of collections within other collections.

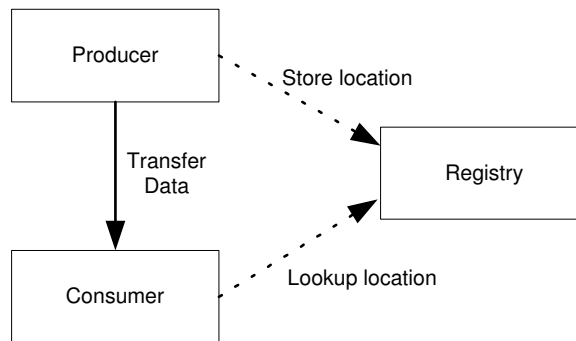
# Chapter 3

## R-GMA

*[The description of the R-GMA contained in this chapter is an updated extract from a paper (see [6]) of which I was a named author]*

### 3.1 THE GRID MONITORING ARCHITECTURE

The Grid Monitoring Architecture (GMA)[4] of the GGF, as shown in Figure 3.1, consists of three components: *Consumers*, *Producers* and a directory service, which is referred to in the R-GMA as a *Registry*.



**Figure 3.1:** 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 3.1 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 [8]; 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.

## 3.2 THE R-GMA

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 [9].

### 3.2.1 Query types and Producer types

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 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. For more detail see Section 5.2.

Three kinds of query are supported: History, Latest and Continuous. The history query might be seen as the more traditional one, where you want to make a query over some time period, including ‘all time’. The latest query is used to find the current value and a continuous query provides the client with all results matching the query as they are published. A continuous query is therefore acting as a filter on a published stream of data.

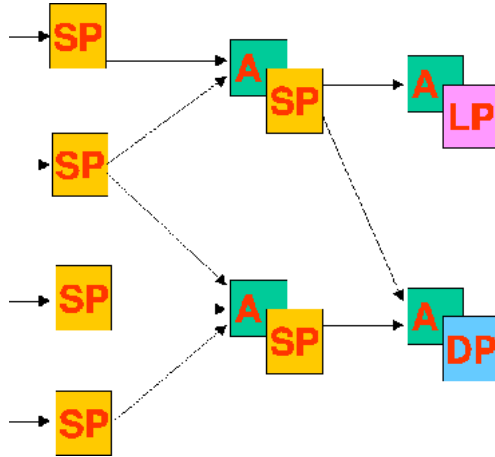
The DataBaseProducer supports history queries. It writes each record to an RDBMS. This is slow (compared to a StreamProducer) but it can handle joins. The StreamProducer supports continuous queries and writes information to a memory structure where it can be picked up by a Consumer. The LatestProducer supports latest queries by holding only the latest records in an RDBMS.

Each record has a time stamp, one or more fields which define what is being measured (e.g. a hostname), and one or more fields which are the measurement (e.g. the 1 minute CPU load average). The time stamp and the defining fields are close to being a primary key, but as there is no way of knowing who is publishing what across the grid, the concept of primary key (as something globally unique) makes no sense. The LatestProducer will replace an earlier record having the same defining fields, as long as the time stamp on the new record is more recent, or the same as the old one.

Producers, especially those using an RDBMS, may need cleaning from time to time. R-GMA provides a mechanism to specify those records of a table to delete by means of a user-specified SQL **WHERE** clause which is executed at intervals that are also specified by the user. For example it might delete records more than a week old, or it may only hold the newest one hundred rows, or it might just keep one record from each day.

Another valuable R-GMA component is the Archiver which is a combined Consumer-Producer and probably should have been called a republisher. You just have to tell an Archiver what to collect and it does so on your behalf. An Archiver works by taking over control of an existing Producer and instantiating a Consumer for each table it is asked to archive. This Consumer then connects to all suitable Producers and data starts streaming from those Producers, through the Archiver and into the new Producer. The inputs to an Archiver are always streams from a StreamProducer. It will re-publish to any kind of **Insertable**. This allows useful topologies of components to be constructed such as the one shown in Figure 3.2, which shows a number of StreamProducers (labelled SP) and a layer of Archivers (A) publishing information via other StreamProducers. Finally there is an Archiver to a LatestProducer (LP) and an Archiver to a DataBaseProducer (DP) to answer both Latest and History queries.

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 database. Currently views have the form:



**Figure 3.2:** A possible topology of R-GMA components

```
SELECT * FROM <table> WHERE <predicate>
```

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

### 3.2.2 Architecture

R-GMA is currently based on servlet technology (although it is currently being converted to web services). Each component has the bulk of its implementation in a servlet. Multiple APIs in Java, C++, C, Python and Perl are available to communicate with the servlets. Figure 3.3 shows the communication between the APIs and the servlets. When a Producer is created its registration details are sent via the Producer Servlet to the Registry (Figure 3.3a). 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 a separate Schema servlet. In practise the Schema is co-located with the Registry. Then when the Producer publishes data, the data are transferred to a local Producer Servlet (Figure 3.3b).

When a Consumer is created its registration details are also sent to the Registry, although this time via a Consumer Servlet (Figure 3.3c). 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 Consumers 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 3.3d-e.

The data is then available to the Consumer on the Consumer Servlet, which should be close in network terms to the Consumer (Figure 3.3f).

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 timestamp is updated in good time.

This relational approach of R-GMA compares favourably to the hierarchical model of LDAP-based systems such as MDS and GridICE. R-GMA allows for a dynamic schema, where users can define and publish their own data, something that is not possible in centrally organised LDAP directory information trees (DIT). R-GMA can provide a global view of the grid by using automatic Registry and Schema replication mechanisms, whereas in LDAP systems hierarchies of intermediaries must be set up manually [9]. It is also necessary to optimise these DITs for popular queries as the LDAP query language is limited. The proprietary Legion system, although not LDAP-based, defines its own query language as opposed to relying on a well-known standard.

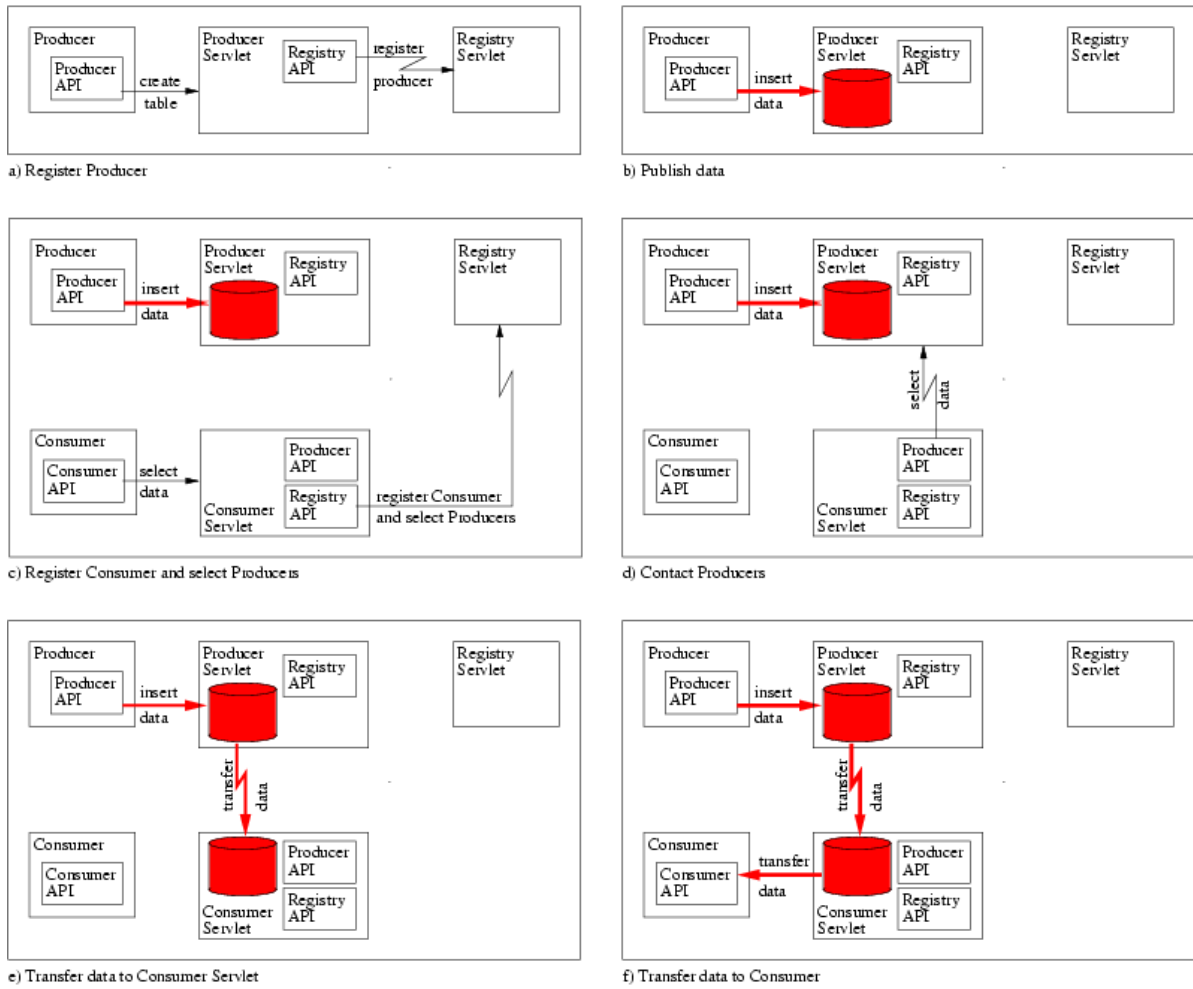


Figure 3.3: Relational Grid Monitoring Architecture

## Chapter 4

# THE DATAGRID AND CROSSGRID PROJECTS

### 4.1 DATAGRID

The main goal of the DataGrid project (led by CERN, the European Organisation for Nuclear Research) was to develop and test a technological infrastructure that would enable the storage, movement and analysis of the huge amounts of data created by the latest class of scientific experiments, such as the Large Hadron Collider (LHC) at CERN. The project considered applications from three areas: high energy physics, led by CERN, biology and medical imaging, led by CNRS France, and earth observations led by the European Space Agency. The project had 21 partners from 11 countries.

The project was organised as four separate working groups, comprised, in total, of 12 workpackages, one of which was responsible for ‘Grid Monitoring’ (WP3). This workpackage was to develop R-GMA, within which the CanonicalProducer was developed as a fundamental component of R-GMA.

## 4.2 CROSSGRID

The CrossGrid project [43] is a major European collaboration involving 21 institutions from 11 different countries. The purpose of the project is to develop applications that allow for the real-time interaction of a person with the application. The example set of applications being developed by the project include interactive simulation and visualisation for surgical procedures, flooding crisis team decision support systems, distributed data analysis in high-energy physics, and air pollution combined with weather forecasting.

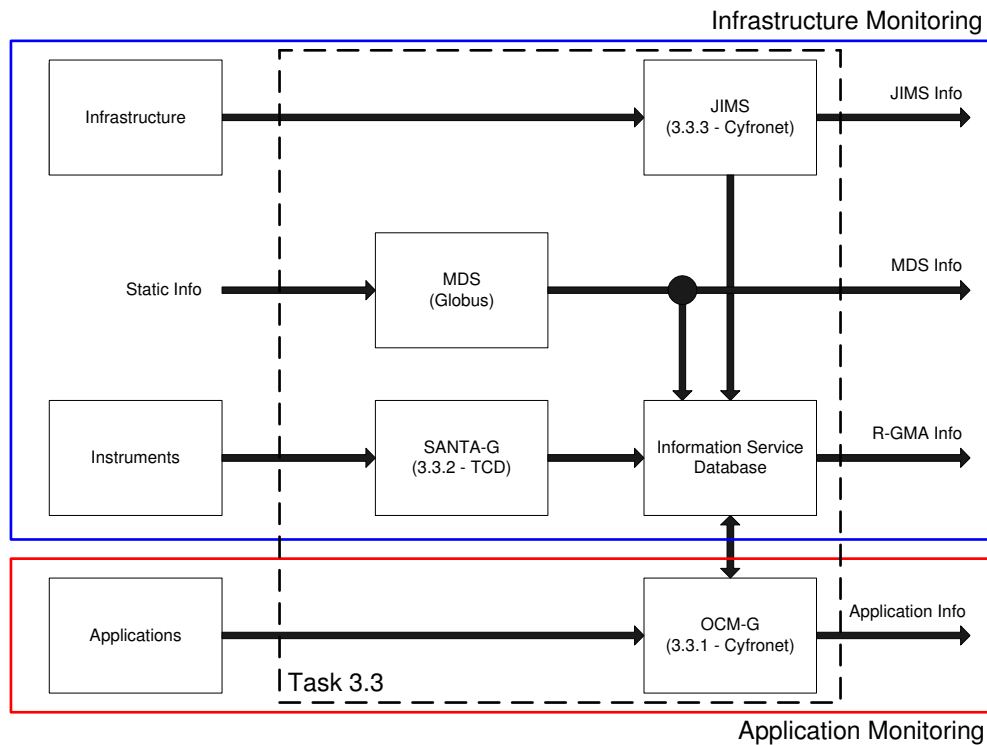
To allow for the development of these applications extensions to the grid environment are required. These include tools for verification of parallel source code, performance prediction, performance evaluation and monitoring. The project was divided into 5 different workpackages, one of which was ‘New Grid Services and Tools’ (WP3), responsible for providing the services and tools required to support the applications and tools developed within the other workpackages.

The NetTracer, referred to within the CrossGrid project as the SANTA-G system, was developed as a part of the ‘Grid Monitoring’ task, a subtask of the ‘New Grid Services and Tools’ workpackage.

The monitoring services developed within the ‘Grid Monitoring’ task, are intended to serve the needs of automatic and interactive performance analysis tools. Their primary function is to deliver low-level data to these tools. During the design and implementation phases the key requirements were system scalability, flexibility and ease of configuration.

For the grid monitoring task it was decided to extend existing grid services (for monitoring instruments and infrastructure), and add new grid services (for applications monitoring), see Figure 4.1.

The *instrument* monitoring services are provided by the monitoring framework and NetTracer demonstrator described in this thesis. These services are a specialized non-invasive complement to other more intrusive monitoring services. The application of these services is



**Figure 4.1:** The grid monitoring system

in the validation and calibration of both intrusive monitoring systems and systemic models, and also for performance analysis.

An *infrastructure* monitoring system that collects static and dynamic information about grid components, such as hosts or network connections, was developed by Cyfronet in Poland. It is based on the Java Management Extensions (JMX) architecture, and is referred to as the JMX-based Infrastructure Monitoring System (JIMS). The information gathered by this system is intended for basic grid activities such as resource allocation or load balancing. Often this type of information has not only immediate, but also historic value. Thus it is often stored in a database for later analysis (e.g. statistical forecasting, etc.). In many ways JIMS replicates the functionality of R-GMA, but was developed as a ‘what if’ alternative to

explore the solution space.

*Application* monitoring was provided by the OCM-G, a grid-enabled version of the OCM, also developed by a group at Cyfronet. OCM-G aims at observing a particular execution of an application. The collected data is useful for tools for application development support. These tools are used to detect bugs, find bottlenecks, or visualize the applications behaviour, and will be most appreciated by software developers.

### **4.3 PROJECT INTERACTIONS**

This thesis describes work that was conducted within both DataGrid and CrossGrid, in a highly productive cross-project collaboration. The CanonicalProducer was developed within DataGrid, while SANTA-G was developed within CrossGrid, both efforts being funded from CrossGrid.

## Chapter 5

# THE SANTA-G MONITORING FRAMEWORK

### 5.1 FRAMEWORK PURPOSE

The purpose of the framework is to provide a generic template to allow for ad-hoc monitoring experiments with external instruments in a grid environment. The framework allows for the information captured by an external instrument to be introduced into a grid information system. Here the information system used is the R-GMA.

The external instruments referred to could be anything. Examples of such devices are logic analysers or oscilloscopes. The difficulty with this class of device is that they generally create a massive amount of data at a very fast rate. To cope with these rates the monitoring data is often stored by the instrument into binary log files. Data stored in this way is not very compatible with the R-GMA model however. When dealing 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.

In order to allow for this a slightly different form of R-GMA producer was necessary, one that did not publish data using the Insertable interface, as with the other producer types, but that allows a user to instead custom-code the way in which the producer responds to a user's request for data. This producer is the CanonicalProducer. The CanonicalProducer is able to cope with large volumes of data by accepting SQL queries and using user-supplied code to return selected information in tabular form when required. The CanonicalProducer forms the central component of the framework.

## 5.2 THE CANONICAL PRODUCER

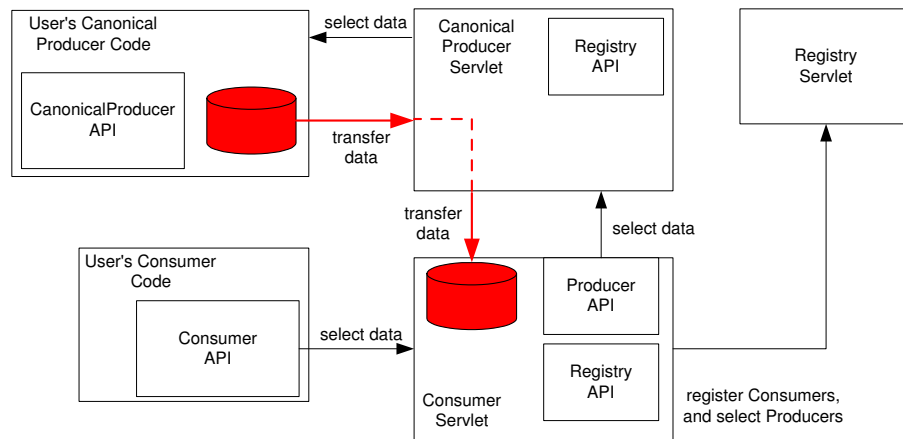
As described in Section 3.2 the R-GMA is built using Java Servlet technology. In order to implement the CanonicalProducer both a servlet and an API had to be conceived.

In general the R-GMA producers are sub-classes of the Insertable class, the class that provides the insert method that 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. Figure 5.1 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.

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 CanonicalProducer servlet. The first parameter is a port number. The CanonicalProducer servlet expects to be able to connect back, by way of a socket connection, to the CanonicalProducer



**Figure 5.1:** CanonicalProducer servlet communication

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",                # table name.
  " WHERE (ipAddress='" + this.ipAddress + "')", # predicate.
  "CREATE TABLE cpuLoadUsage( " + # create table
  "ipAddress VARCHAR(50) NOT NULL PRIMARY KEY, " + # statement.
  "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 functionality, hence its name. 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.

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

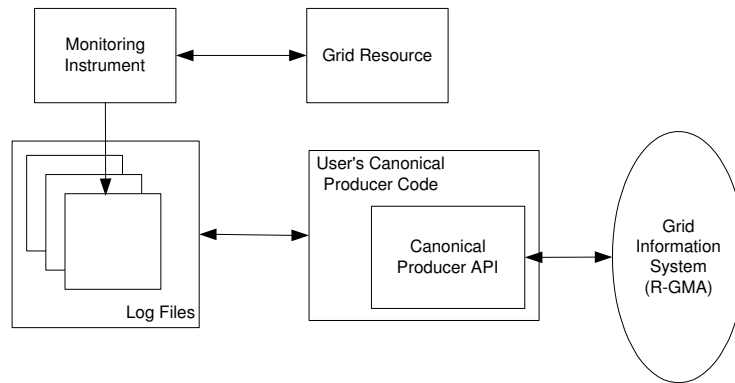
```
<?xml version = '1.0' encoding='UTF-8' "standalone='no'?>
  <edg:XMLResponse xmlns:edg='http://www.edg.org'>
    <XMLResultSet>
      <rowMetaData>
        <colMetaData>ColumnName
      </colMetaData>
    </rowMetaData>
    <row><col>ColumnValue</col></row>
  </XMLResultSet>
</edg:XMLResponse>
```

One 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. Hence, 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.

### 5.3 FRAMEWORK ARCHITECTURE

The framework architecture can be divided into three main parts; the device used to capture the monitoring data from the resource, i.e. the external instrument, the CanonicalProducer code used to access the data, and the CanonicalProducer API used by the code to register with the R-GMA and to declare the tables that the code publishes, see Figure 5.2.

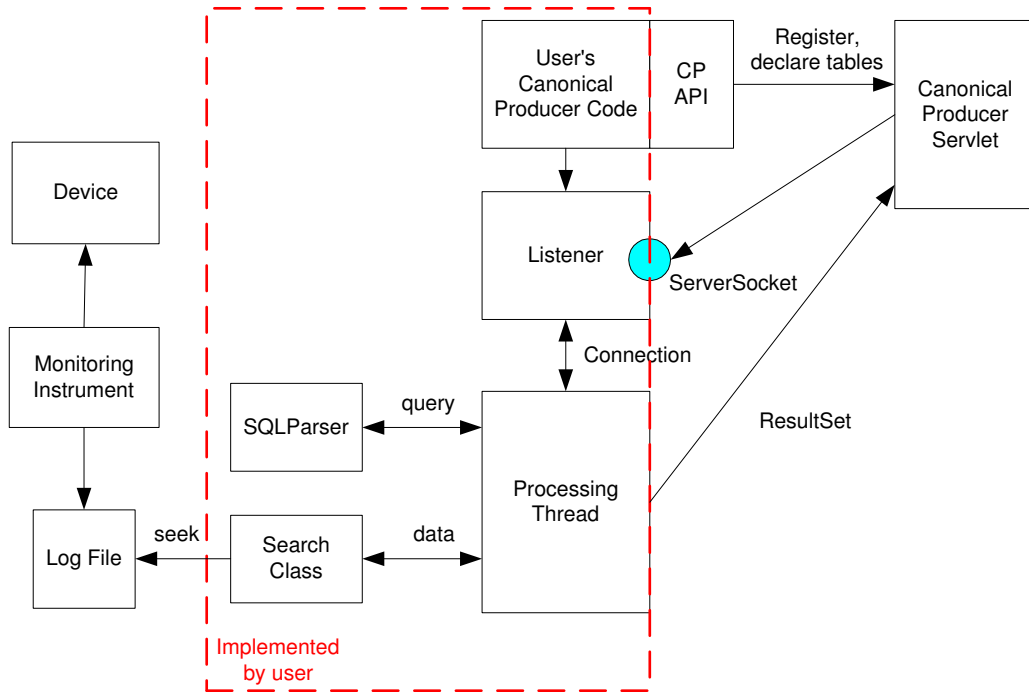


**Figure 5.2:** Monitoring framework

In the framework the CanonicalProducer code provides the bridge between the raw data and the R-GMA system. A typical implementation of the CanonicalProducer code would consist of several components, as shown in Figure 5.3. Although the figure shows the data being collected by an instrument and stored in log files the data source could be anything. For example the code could be used to trigger a script to collect the required data in response to a received query.

**Producer Code:** this 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:** this would be created by the main class. It would need to create a `ServerSocket`



**Figure 5.3:** An example of CanonicalProducer user code

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. The Listener would then continue listening for new connections.

**Processing Thread:** a new 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 can then be returned to the servlet, over the same socket connection.

**SQL Parser:** some additional utility classes would be needed by the processing thread, for example, a class to parse the SQL SELECT statement received from the servlet.

**Search Class:** a class would also be needed to search the available data for results that

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. Obviously many optimizations are possible.

## Chapter 6

# DESIGN AND ARCHITECTURE

The following chapter describes the design and architecture of the NetTracer. The NetTracer implements the framework described in the previous chapter; it was designed by the author, within the EU CrossGrid project, as the demonstrator of this concept. As stated the framework is comprised of three main components:

1. an external monitoring instrument, to obtain the monitoring data
2. the CanonicalProducer API, to interface with the grid information system
3. the user's CanonicalProducer code, to access the monitoring data

The NetTracer was designed to provide three main functions:

1. Allow a user to initiate tracing of grid resources, collect the monitoring data, and provide access to the data through the grid information system
2. Allow a user to select the required subset of data, by way of the grid information system
3. Provide the information required by dependent subsystems within the CrossGrid grid services and tools system

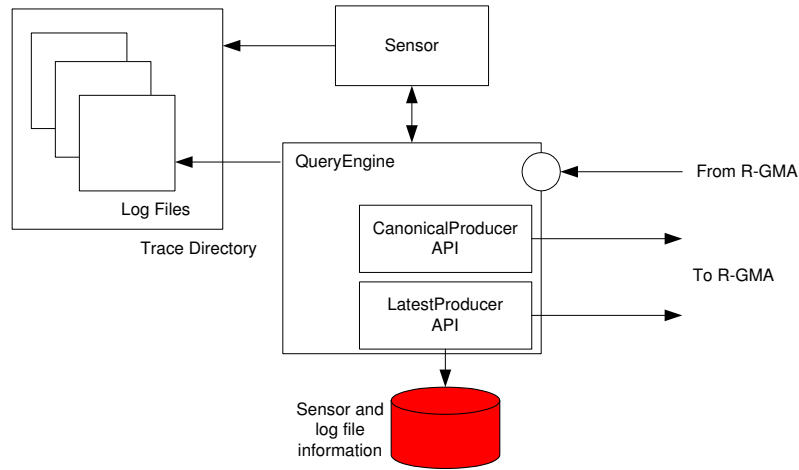
The NetTracer can be broken down into two main modules that provide these functions: a publishing module and a viewer module.

The purpose of the publishing module is to monitor the log files created by the external monitoring instrument, 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 are then presented to users graphically or in a table.

The publishing module is composed of a further two components, the QueryEngine and the Sensor, as can be seen in Figure 6.1. 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(s) 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.

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 6.2.

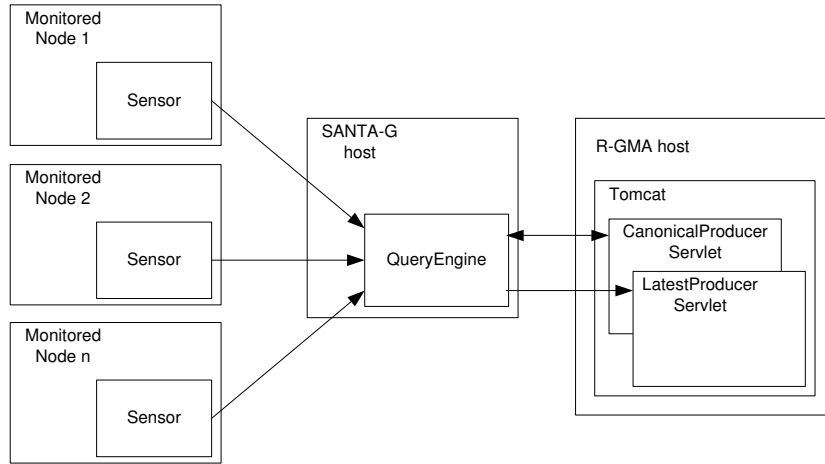
The sensor, in this demonstrator, as it will run on the node to be monitored, must be as lightweight and as easy to install as possible. For this reason the majority of the processing and data storage should be handled by the QueryEngine. In order for a user to be able to query the NetTracer they must be able to obtain information about the available sensors



**Figure 6.1:** NetTracer publishing module

and the log files stored on the nodes hosting these sensors. The original design was for the sensor to maintain tables of its log files and to publish these by way of a R-GMA DataBase Producer. This was considered too intrusive, however, as to run a producer the R-GMA would have to be installed on every monitored node. To avoid this, the task of storing the file information was moved to the QueryEngine. The situation became analogous then 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.

This solved a second issue with the initial design. In the first design the QueryEngine was not aware of the currently running sensors or the files they held. In order for a user to submit a query to a NetTracer they first had to query the R-GMA to obtain details of the available sensors and log files, and then use this information when submitting a query to the QueryEngine. The design was changed so that sensors now register with the QueryEngine on start-up. The QueryEngine records the sensor details and publishes them to the R-GMA. Also the sensor sends updates to the QueryEngine when new log files are detected, which are again stored and published to the R-GMA by the QueryEngine. If a sensor is shutdown the



**Figure 6.2:** Example NetTracer deployment

QueryEngine is informed and its details are removed. The detailed reasons for this change are covered in Section 7.1.2.

The sensor provides for two example data sources, Tcpdump and Snort. Tcpdump captures network packets from a node’s network interface card and stores the packet data into raw binary log files. Snort is a network intrusion detection system that can log alerts, and the packet that triggered the alert, to a Tcpdump-compatible logfile, when suspect packets are detected.

The QueryEngine is the central component of the publishing module. It is the QueryEngine that provides the interface from the NetTracer to the R-GMA by using the CanonicalProducer API. It implements the components described in Section 5.3. It is the QueryEngine that receives queries submitted to the R-GMA. It 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 QueryEngine is designed to be extensible, i.e. the core functionality, the R-GMA interface, query parsing, etc., can be re-used for multiple data sources. This is achieved through the use of an abstract Filter class and a XML schema file. The Filter class can be

extended to allow for data stored in different log file formats. The Filter class describes the required methods for accessing the log files, loading a packet contained in the log file, and obtaining fields from the packet.

```
abstract class Filter {  
    /**  
     * Tests to see if the currently loaded packet occupies  
     * a row in the table identified by tableId.  
     */  
    abstract boolean checkTableConstraints(int tableId);  
  
    /**  
     * Closes the current log file.  
     */  
    abstract void closeFile () throws IOException;  
  
    /**  
     * Translates a value given in a WHERE predicate into a value that can  
     * be used during query execution.  
     */  
    abstract Object decodeWhereValue(int tableId , int fieldId , String toDecode);  
  
    /**  
     * Extracts a field from the currently loaded packet  
     * for inclusion in a result set.  
     */  
    abstract String getResultField(int tableId , int fieldId);  
  
    /**  
     * Extracts a field from the currently loaded packet  
     * for comparison with a WHERE predicate value.  
     */  
    abstract Object getSearchField(int tableId , int fieldId);  
  
    /**  
     * Loads a log file.  
     */  
}
```

```

abstract int loadFile(RemoteFile file);

/**
 * Loads the next network packet from the current log file.
 */
abstract int loadNextPacket();
}

```

In order to create a new QueryEngine for a specific log file format a new Filter must be written that implements each of these methods. The Filter class is used to access a log file, by using the `loadFile()` method, and to read network packets from this file, using the `loadNextPacket()` method. The tables of data that the QueryEngine is to publish are described in a XML schema file. This file is parsed by the QueryEngine on startup, and the required ‘CREATE TABLE’ statements needed to register the tables in the R-GMA are generated. Each table is described in the file in the following way:

```

<table id="0" name="TableName">
  <field id="-4" key="primary" type="VARCHAR(100)">siteId</field>
  <field id="-3" key="primary" type="VARCHAR(100)">sensorId</field>
  <field id="-2" key="primary" type="INT">fileId</field>
  <field id="-1" key="primary" type="INT">packetId</field>
  <field id="0" type="VARCHAR(100)">column1</field>
  <field id="1" type="VARCHAR(100)">column2</field>
  .
  .
  .
</table>

```

Each table, and each column within the table, is assigned a unique ID and a name. The 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, `fileId`, which identifies a particular log file on a sensor node, and `packetId`, which identifies a packet within the log file, are required in every table. 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 and published in each table as separate second and microsecond fields.

The `QueryEngine` when parsing a SQL query translates fields into `(tableId, fieldId)` pairs, as defined by the schema. These values are then used in the `Filter` class to obtain the required field from the currently loaded network packet. Two methods must be implemented for this, `getSearchField()` and `getResultField()`. `getSearchField()` should read a field from the packet in a form that can be used in comparisons during query execution, for example, in the `EthernetFilter`, fields are read from the `Tcpdump` log file as `long` values. `getResultField()` obtains the values as they should appear in a result set, for example, again with the `EthernetFilter`, IP addresses read from the file as `long` values are converted by this method to IP addresses in the form `iii.jjj.kkk.lll`. The final method `decodeWhereValue()` is necessary when values specified as part of a `WHERE` predicate need to be converted prior to query execution. For example, if searching for a specific IP address, the IP address must be first converted by this method into the same data type as is returned by the `getSearchField()` method.

Two example filters and their associated XML schema files have been implemented, an `EthernetFilter` for use with `Tcpdump` format logfiles (i.e. Ethernet tracing), and a `SCIFilter` for use with trace files collected by the Computer Architecture Group's SCI [1] trace instrument [24].

For the demonstrator, 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 a 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.

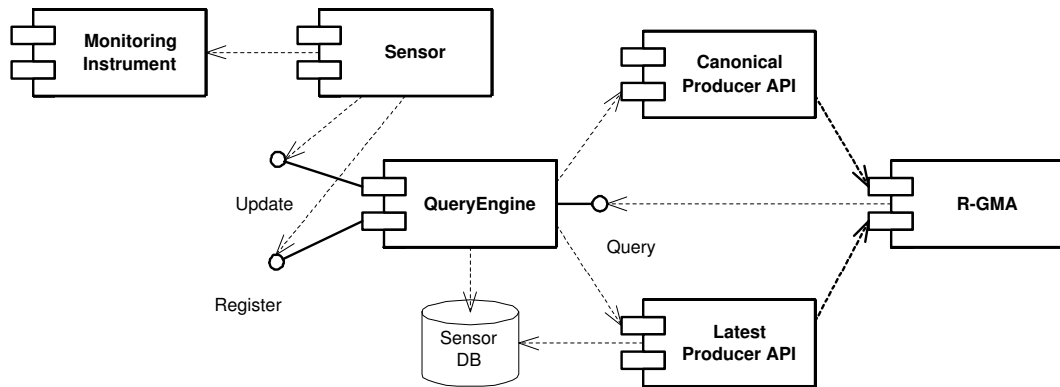
The design of the Viewer module is reasonably straightforward. A Java Swing GUI is provided which makes use of the Consumer API to submit SQL queries to the R-GMA and to receive result sets in response. The individual fields of these result sets are then extracted and displayed either graphically or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

# Chapter 7

## IMPLEMENTATION

### 7.1 THE PUBLISHING MODULE

The following component diagram, Figure 7.1, shows a UML representation of the structure of the publishing module as it was implemented.



**Figure 7.1:** Publishing module implementation structure

### 7.1.1 The Sensor

In the NetTracer system, for network monitoring, two main sensor types have been implemented, `Tcpdump` and `Snort`. The `Snort` sensor type is described in detail in Chapter 10. A third sensor type, the `Static` sensor, is used to publish details of a set of static pre-acquired log files, principally for testing purposes. The startup and shutdown sequence for all three types is the same. It is the functionality of the sensors whilst they are running that distinguishes the sensor types.

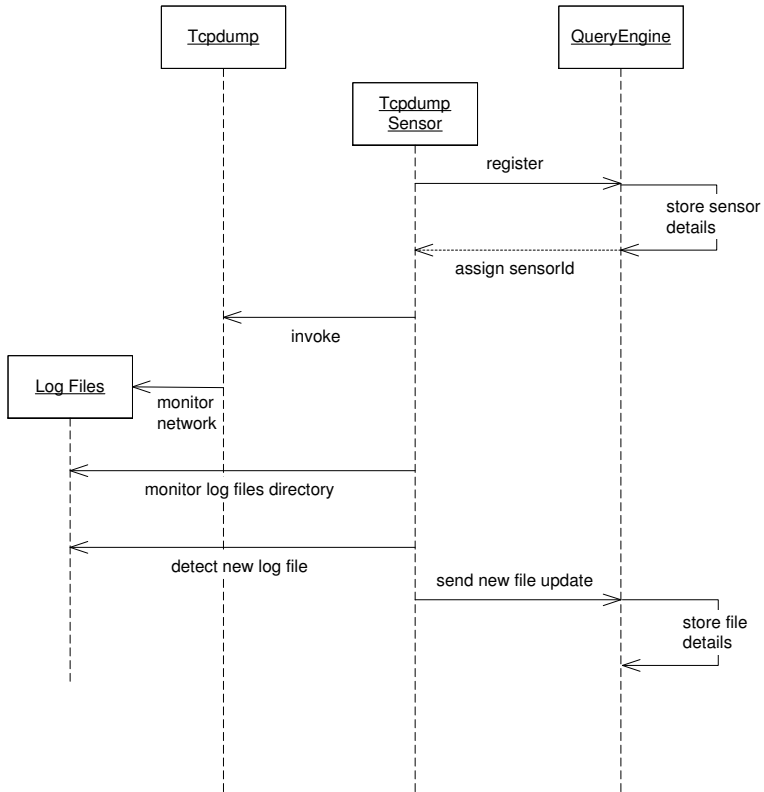
A sensor's configuration is stored in a file, generated by a configuration script. This file contains the type of sensor to start (i.e. `static`, `tcpdump` or `snort`), contact information for the `QueryEngine` (i.e. hostname and port number), the DN of the authorised `QueryEngine`'s host certificate, and the trace directory (i.e. the directory that contains the log files), as well as sensor type specific information.

A sensor registers with the specified `QueryEngine` at startup, and upon successful registration receives an ID in response, which is then used in all future communications with the `QueryEngine`. A sensor unregisters at shutdown by sending a message to the `QueryEngine` informing it that the sensor is closing, and that its details should be removed from the sensor information tables.

The `Tcpdump` sensor type is used to both invoke `Tcpdump`, to collect network traffic, and to then monitor the log files of network packet data generated. The sensor invokes `Tcpdump`, using the arguments contained in the sensor's configuration file. These arguments specify the directory into which `Tcpdump` should write (i.e. the trace directory), and also the maximum size of a log file. When a log file reaches this maximum size, `Tcpdump` closes the file and begins writing to a new log file. The sensor monitors the trace directory, waiting for new log files to be created. Upon detection of a new log file the sensor informs the `QueryEngine`, which records this event. The sensor maintains the log files in a queue, the maximum length of which is stored in the sensor's configuration. When a new file is detected it is added by the

sensor to the head of the queue, and if by so doing the maximum queue length is exceeded, the oldest file is removed and either deleted or archived, i.e. compressed and moved to an archive directory.

Figure 7.2 shows the startup sequence for a Tcpdump sensor.



**Figure 7.2:** Tcpdump Sensor startup sequence

It was intended to keep the sensor as simple and lightweight as possible. The structure of the sensor component is shown in Figure 7.3. The following describes the classes and their function:

**Sensor:** the main class of the sensor component.

**TCPdumpInvoker:** used by the Tcpdump sensor to invoke a Tcpdump process. Tcpdump

is started using the arguments specified during configuration of the sensor. Also specified in the sensor's configuration are the `trace directory`, the directory into which Tcpcmdump will create the log files, and the `maximum log file size`, the size at which Tcpcmdump will close the current log file and create a new file. The filename is composed from the hostname of the machine followed by a number, which is increased with each file created, e.g. `hostname`, `hostname2`, `hostname3`.

**FileMonitor:** used by the Tcpcmdump sensor to periodically poll 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 removed and passed to the ArchiveFile class. If the archive variable is set to ARCHIVE in the sensor's configuration, then this class will compress the file and move it to the archive directory, otherwise the file is deleted.

**SnortMonitor:** used by the Snort sensor type to monitor the Snort alerts log file. When a new alert is entered into the file, the SnortMonitor reads the alert, and sends an update message that encapsulates the alert to the QueryEngine.

**SnortFileMonitor:** monitors the packet log files generated by Snort. Updates are sent to the QueryEngine when new log files are detected.

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

The Sensor component also provides access to the log files stored in the trace directory on the host machine, by implementing a simple file server that listens for file access requests from the QueryEngine. A class diagram for the file server is shown in Figure 7.4(a).

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 EDG Java security package [45] in order to validate the certificates presented to it by a client during the SSL handshake. The client's DN is checked to ensure that it matches that of the trusted QueryEngine, which is stored in the sensor's configuration file.

The *ClientThread* class is responsible for serving the requested file to the client. It reads the filename from the socket and accesses the requested file. The required number of bytes are read from the file by the *ClientThread*, and sent back to the client over the SSL connection.

The sensor's file server is also used to solve an issue that arose with the implementation of the log file queue mechanism described above. When the log file queue is full and a new file is added to the head of the queue, the oldest file is removed (and possibly deleted), and the ID's of the remaining files are updated to reflect this. Should this occur during the execution of a query the results would be invalidated, as the query would now be carried out on a different file, or possibly on a file which was just deleted. To avoid this a file locking mechanism was added. Upon execution of a query the files currently stored in the sensor's queue are locked by sending a `lock` request to the sensor's file server. The sensor is prevented from deleting a locked file if a queue change occurs, until the file is unlocked upon query completion. A timeout is used to prevent files remaining locked indefinitely should a query fail.

Figure 7.4(b) shows the client side of the remote file server. This 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 a *RemoteFile* object is created and passed to the *RemoteFileInputStream*. The *RemoteFileInputStream* is then used in the same way as the standard *FileInputStream* to read the remote file. When instantiated the *RemoteFileInputStream* attempts to connect to the file server running on the remote host by creating an *SSLSocket*, again using the EDG 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 file is returned. Subsequent calls to read the file now read in from the socket connection.

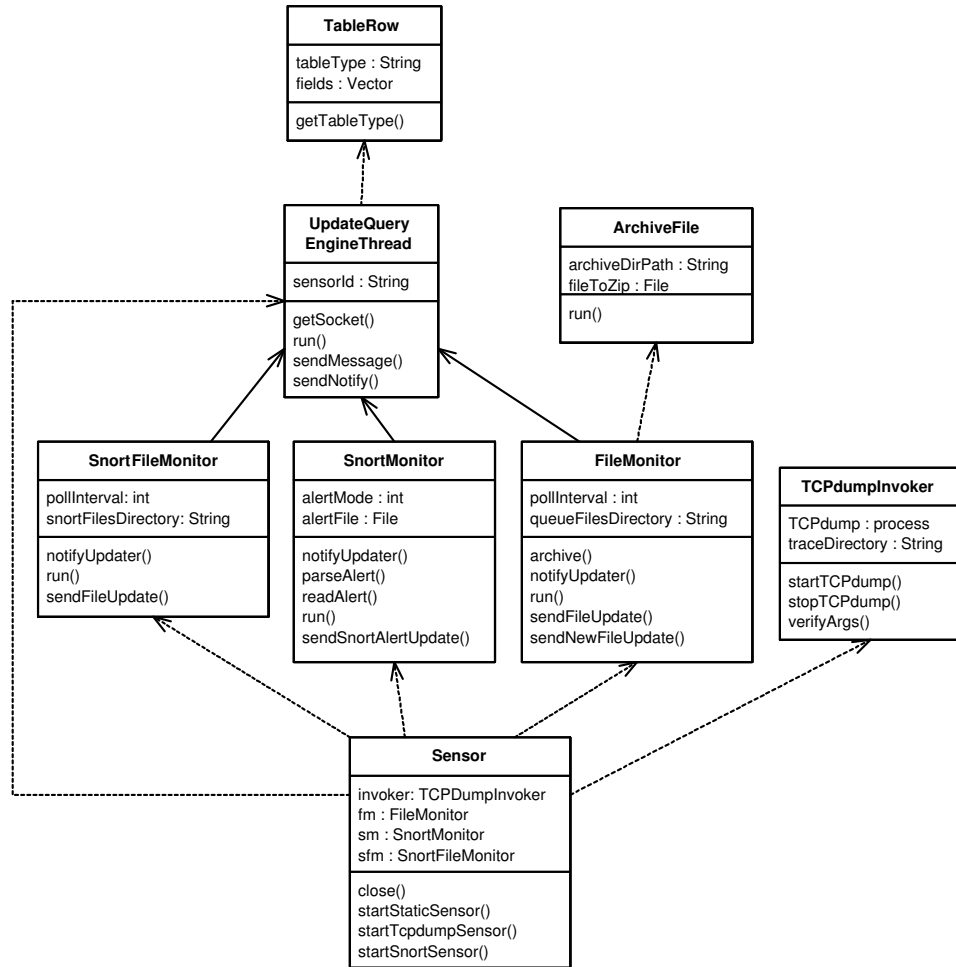
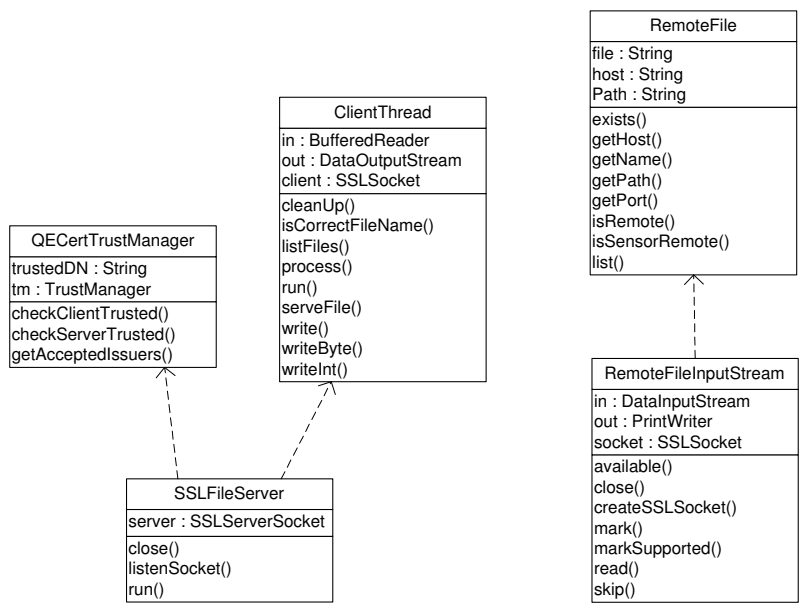


Figure 7.3: Sensor class diagram



(a) Server

(b) Client

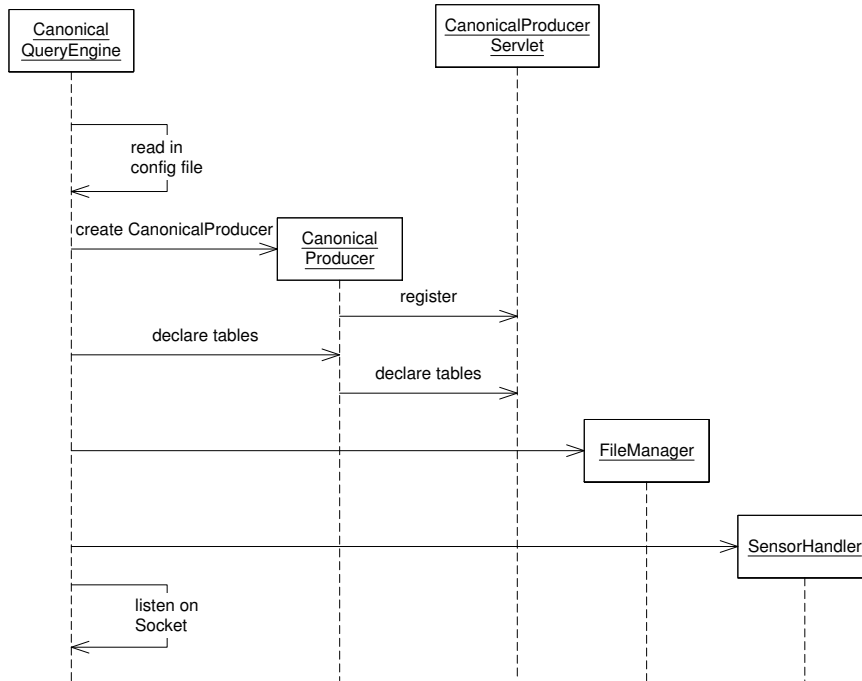
**Figure 7.4:** Remote file server

### 7.1.2 The QueryEngine

The QueryEngine has three main functions: it provides the interface to the R-GMA through the CanonicalProducer API, it executes 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 uses the CanonicalProducer object to declare the tables of information that it provides. The QueryEngine also maintains a set of sensor information tables. There are two main sensor information tables, `sensors` and `sensorFiles`. `Sensors` stores the basic sensor information, such as the sensor's ID, type, host, etc. The `sensorFiles` table holds information on the log files currently stored on the sensor's node, the file ID, the file name, etc. An extra table, `snortAlerts`, is used to store and publish the alerts detected by a Snort sensor (see Chapter 10). 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 7.5. 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 7.6 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 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*.



**Figure 7.5:** QueryEngine startup sequence

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.

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 for ethernet data from a Tcpdump format log file:

```
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, 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. An offset into the file is calculated for each of the fields needed. The size of each field in bytes is 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 is composed of several classes, as shown in the class diagram, Figure 7.7.

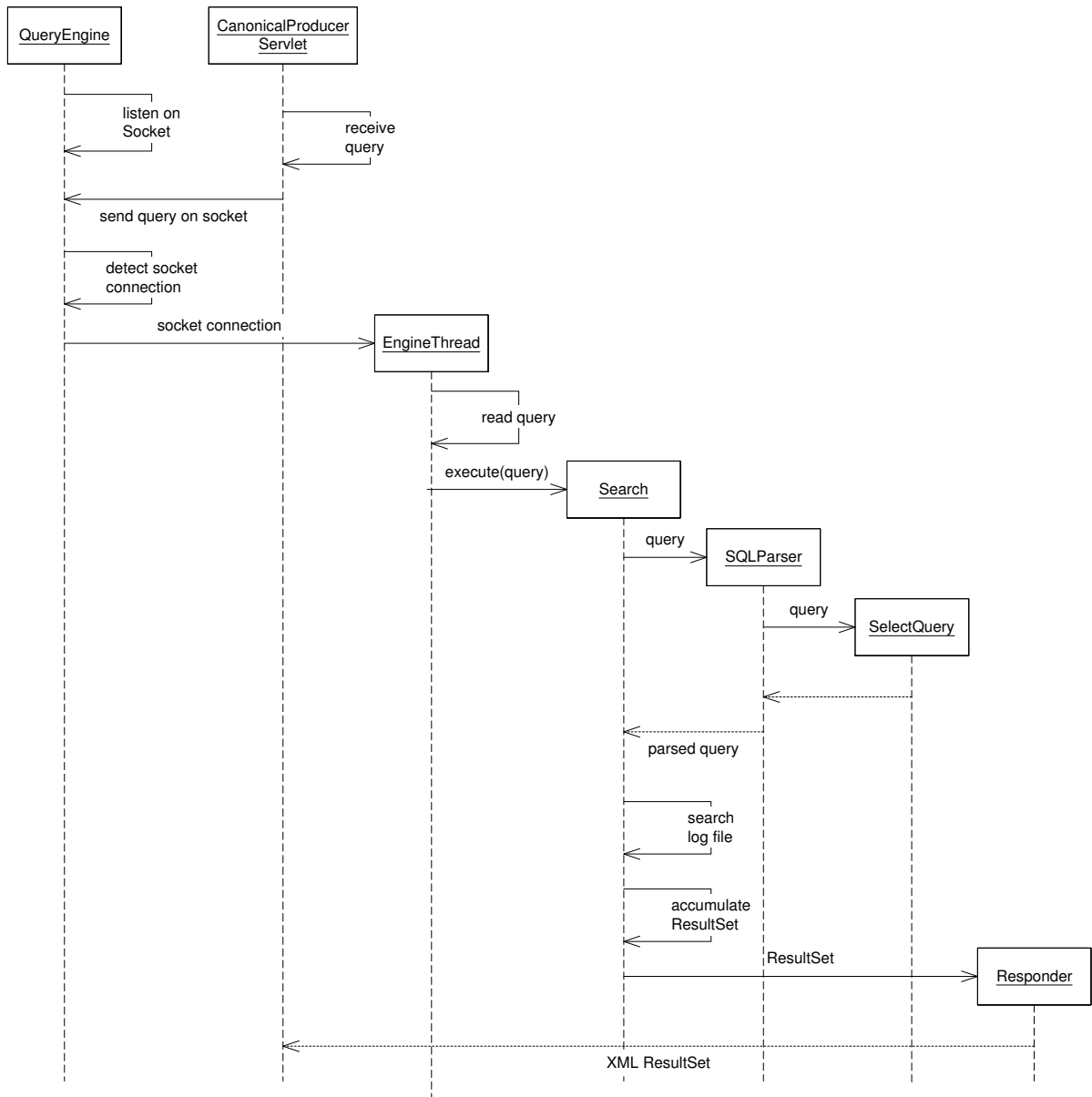
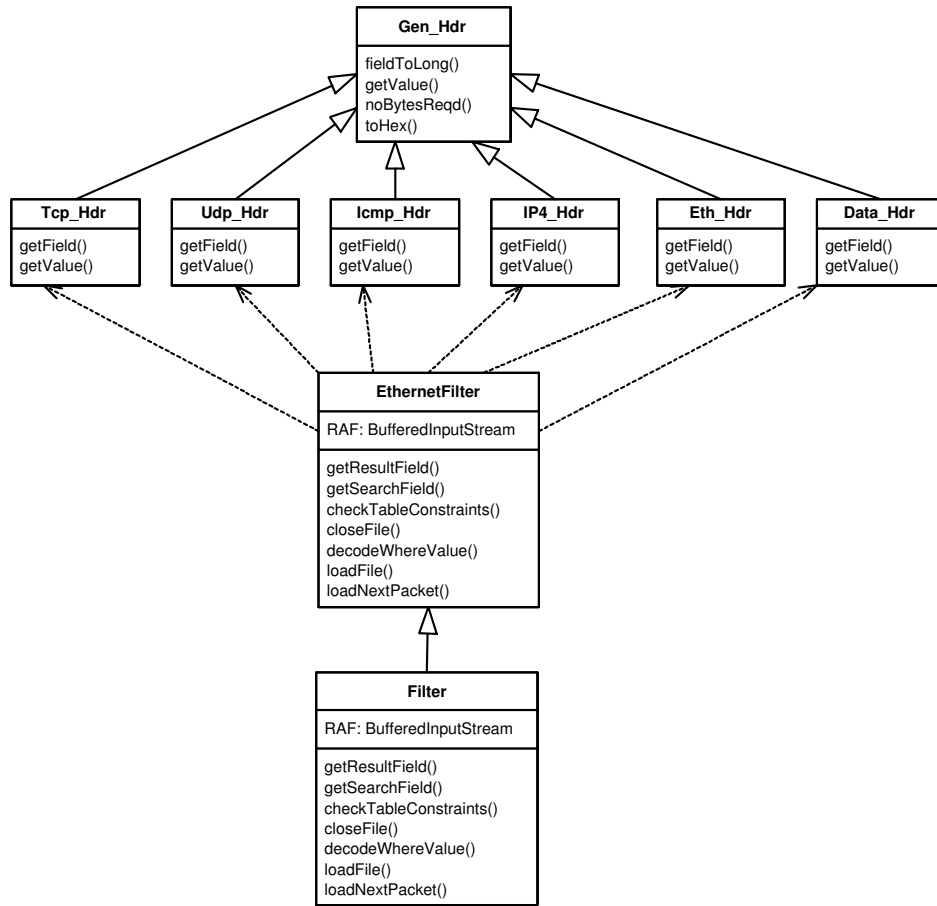


Figure 7.6: QueryEngine query processing sequence



Two implementations of the QueryEngine for different network interconnect technologies have been provided, ethernet and SCI. Ethernet packets can be divided into a series of network protocols. Each layer in the network protocol appends its own header to the packet. The ethernet schema was designed therefore so that each table in the schema 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 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. This schema was described in an XML file and an *EthernetFilter* class was created. This class accesses the Tcpdump format log files and loads network packets from them. Figure 7.8 shows the structure of the EthernetFilter.

A similar approach was taken for the *SCIFilter*. The SCI QueryEngine provides an example of using the SANTA-G framework with a hardware instrument. The SCI trace instrument allows for non-invasive deep tracing of SCI interconnect traffic. The instrument is connected via trace probes to the output link of an SCI interconnect. SCI packets traced from the target node are stored by the instrument to a text format log file. It is from this log file that the *SCIFilter* loads the raw packet data. With the original SANTA tools a second stage of instrumentation was used to decode the trace file to separate database table files, suitable



**Figure 7.8:** EthernetFilter class diagram

for importation to a RDBMS, each table representing a field in the SCI packet header. The database schema has been retained in the *SCIFilter*, having been translated to the XML format required by the QueryEngine. Figure 7.9 shows the structure of the *SCIFilter*.

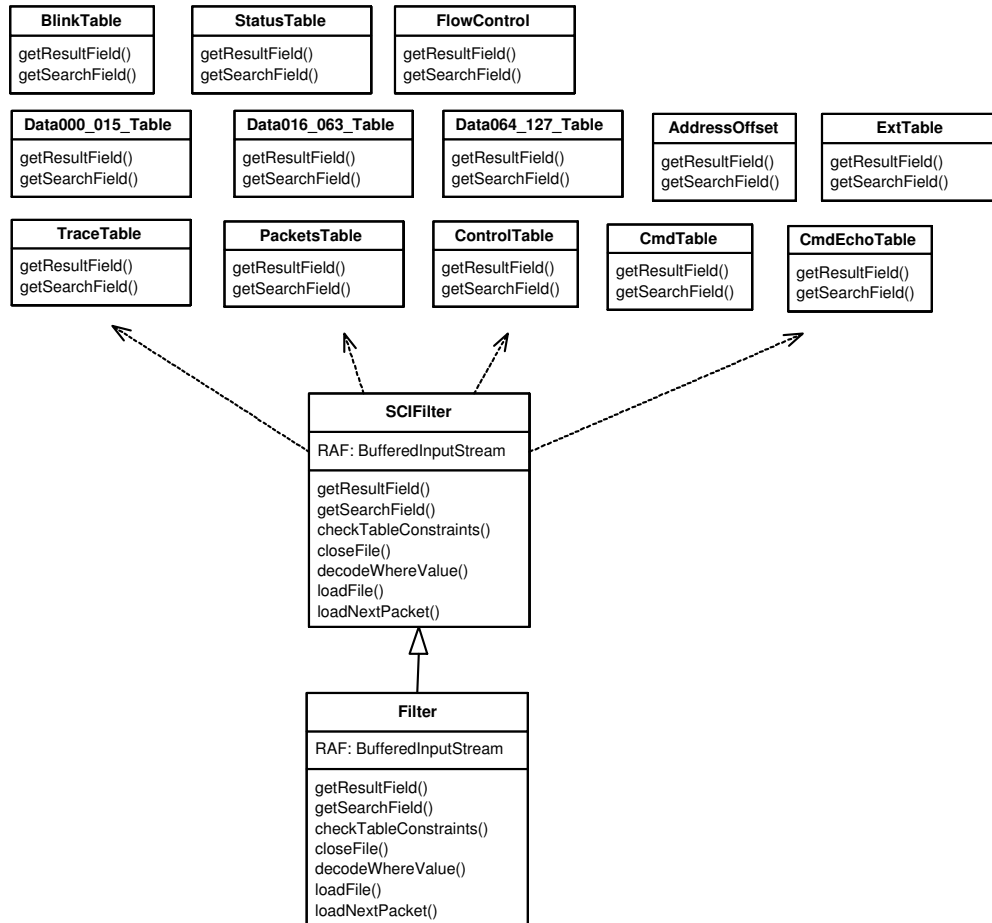


Figure 7.9: SCIFilter class diagram

The QueryEngine must also process messages received from sensors. When the QueryEngine reads a message from a socket connection the message is parsed to see if it is a 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 a new sensor message is sent to the QueryEngine, `newsensor;sensorhost;sensortype`, where *sensorhost* is the hostname of the machine hosting the sensor, and *sensortype* is the type of sensor. 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 Latest Producer. The QueryEngine stores two tables of information relating to sensors, as described above. Through these tables users can determine how many sensors are currently running, and the number of log files currently stored by them.

In the first NetTracer implementation an independent Database Producer was used to store the sensor information. There were a number of reasons for choosing to alter this. In order to use the Database Producer, a MySQL database had first to be created for the producer to use. This complicated the configuration of the NetTracer for users. In order to configure the system a user required an account on the MySQL RDBMS with create privileges. A setup script was provided that created the required sensor database, however, this would not work if it was executed on a machine that did not have remote access to the MySQL database. Also a Database Producer is intended to be used to persistently store the history of a published stream of information for historical analysis. The sensor information stored by the QueryEngine is dynamic, and of no historical significance. Furthermore, in order to update the information, or to delete information when a sensor was closed, it was necessary to open a direct connection the database, as the R-GMA would not except ‘DELETE’ or ‘UPDATE’ SQL statements. The R-GMA installation guide, however, recommends that remote management of the MySQL database should not be allowed, and that the port for

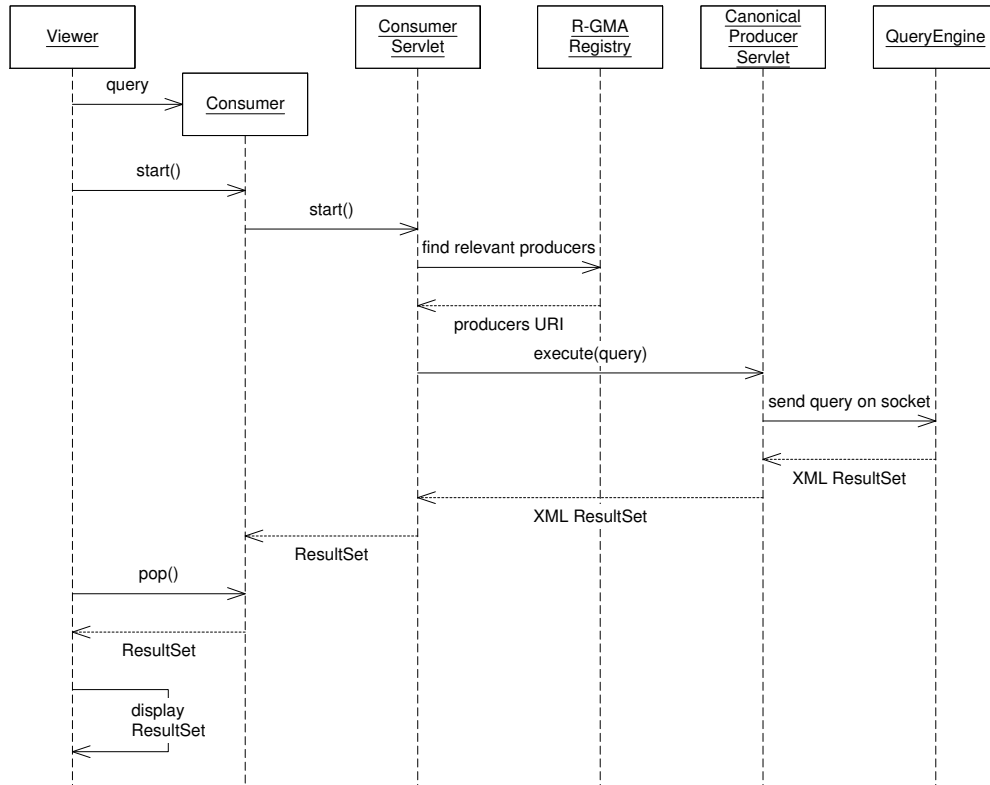
this be blocked by a firewall. For these reasons the implementation was altered to use a Latest Producer bound to the QueryEngine. A Latest Producer only stores the most recent tuple for a given primary key. When a tuple is inserted with the same primary key as an existing tuple, the older tuple is replaced. This simplifies updates of the information, as all that is required is to insert the new row, and the old row will be removed automatically. In order to remove information on closed sensors the Latest Producer's clean-up predicate mechanism was used. This allows for a producer to be created that will periodically remove the rows that match the clean-up predicate from its published tables. For the sensor tables an additional column, `status` was added, which indicates whether a sensor is currently running or not. A clean-up predicate is specified as a WHERE clause, so to remove stopped sensors the clean-up predicate specified was 'WHERE status = "closed"'. When a sensor is shutdown the QueryEngine updates the sensor's status to 'closed'. When the clean-up thread is executed any row in which the status is set to 'closed' is removed.

### 7.1.3 The Viewer module

The Viewer provides a Java Swing GUI that allows users to collect and view data published by the NetTracer. The Viewer GUI has two main panels, the packet view and the query panel. The packet view displays a packet from the selected log file. Currently only the ethernet QueryEngine is supported. The query panel allows a user to submit a 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

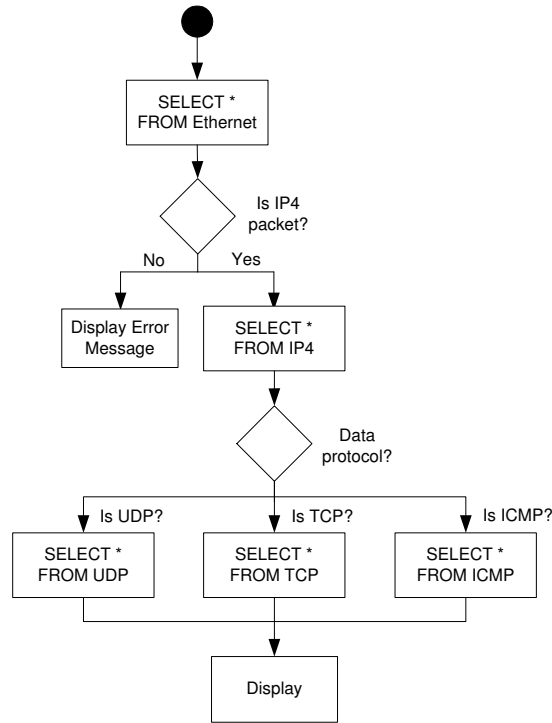
the form of a result set. The sequences of events that occur when a query is submitted from the Viewer are shown in Figure 7.10.



**Figure 7.10:** Viewer query submission sequence

In order to display an ethernet packet the packet's type must first be determined by querying the Ethernet table (see Figure 7.11), which contains the packet type field. The type can then be decoded. Currently only IP version 4 packets are supported. If this type of packet is detected then the IP4 table can be queried to obtain the IP header information for the packet. From this result set we can obtain the data protocol of the packet. Currently TCP, UDP, and ICMP type packets are supported. The rest of the packet data can then be obtained by querying the required data protocol table. Once this is done the display is constructed for the type of packet found. Figure 7.12 shows the packet view panel displaying

a TCP packet.



**Figure 7.11:** Ethernet packet display process

The query panel is quite straightforward (see Figure 7.13). A text area allows the users to enter a SQL query. The query is submitted to the R-GMA by pressing the **execute** button. When the query completes and a result set is returned to the Viewer, the individual fields are extracted and displayed in a table under the text area. The full packet from which the fields in a row are taken can then be displayed in the packet view by double clicking a row in the table. The table resulting from a SQL query can also be printed by clicking the **print** button.

A number of additional windows can be opened from the query panel. A query builder allows the user to construct, save, and load SQL SELECT statements (see Figure 7.15). A sensor information panel provides a summary of the currently running sensors (see Fig-

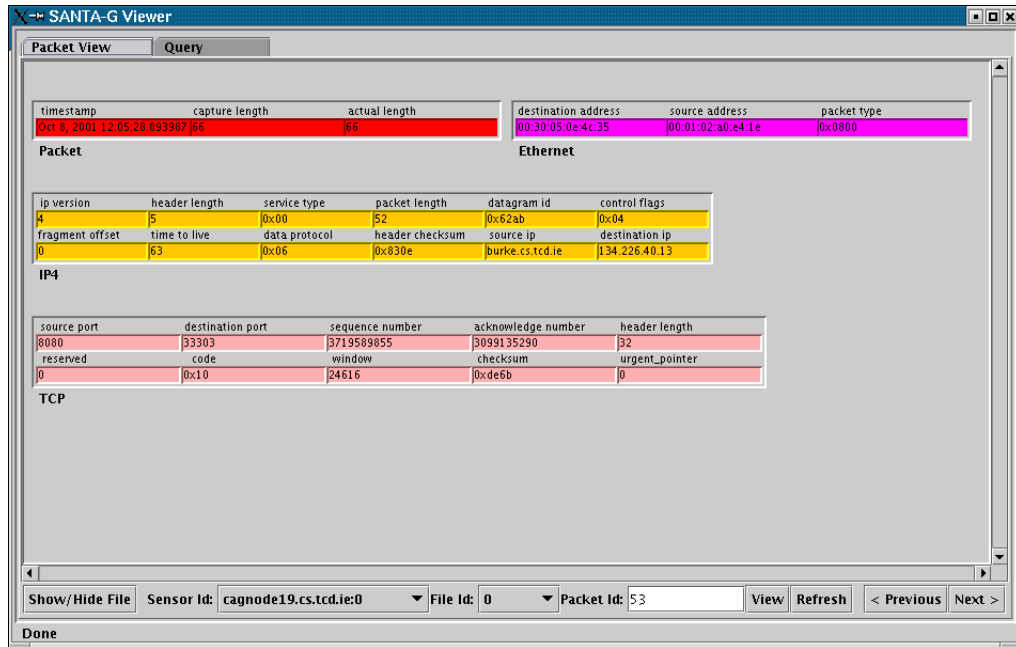


Figure 7.12: Viewer GUI, packet view panel

ure 7.14). By clicking on a particular sensor the full information on that sensor can be viewed, such as the sensor’s host and type, and the log files currently stored. The Snort alerts panel displays any alerts that have been logged to the snortAlerts table by a Snort sensor (see Figure 7.16). An alert can be selected and either the full text of the alert, as it would appear in the Snort alert file, or the full header data of the packet that triggered the alert can then be viewed. Very obviously this can be extended; this is just a demonstrator.

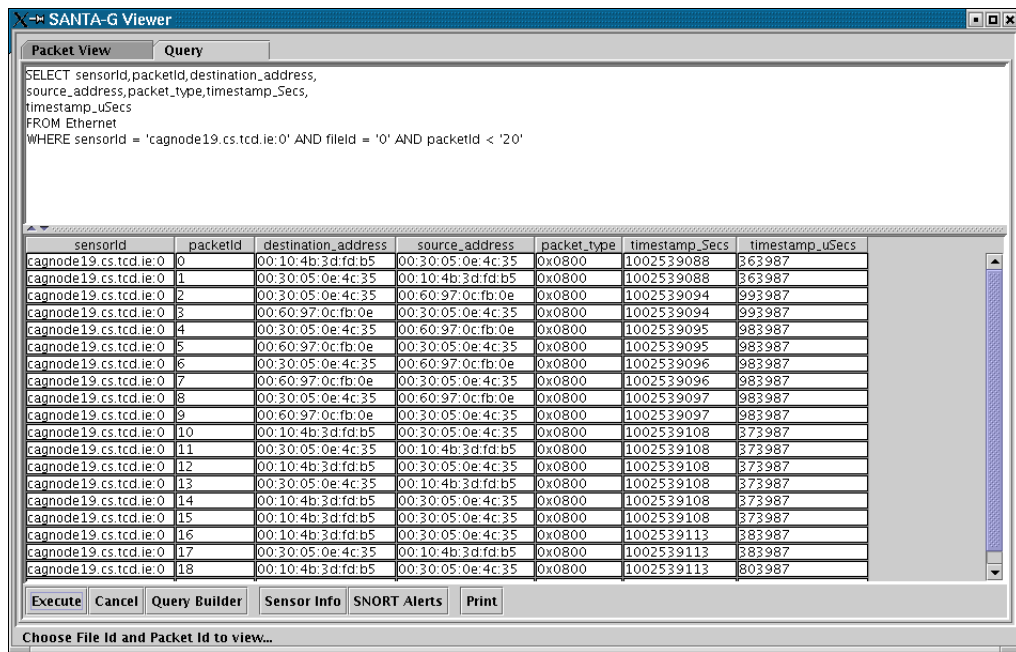


Figure 7.13: Viewer GUI, query view panel

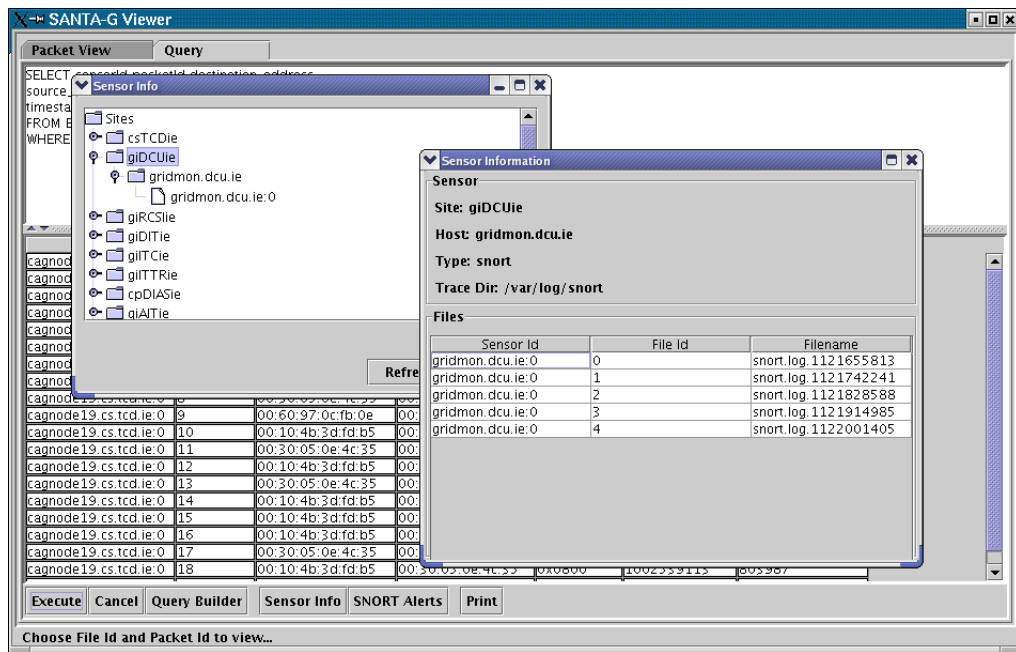


Figure 7.14: Viewer GUI, sensor information panels

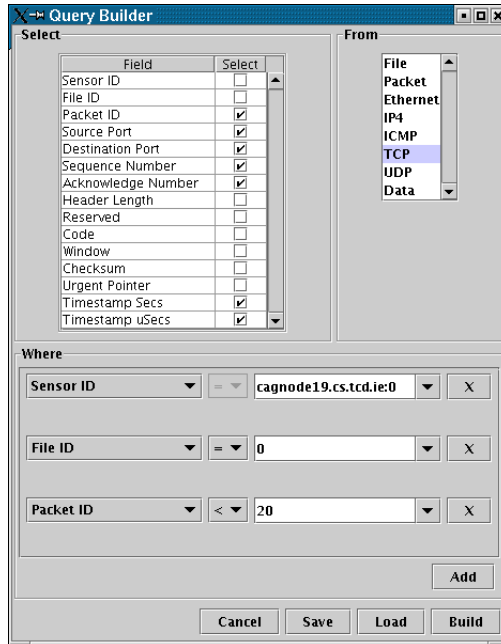


Figure 7.15: Viewer GUI, query builder

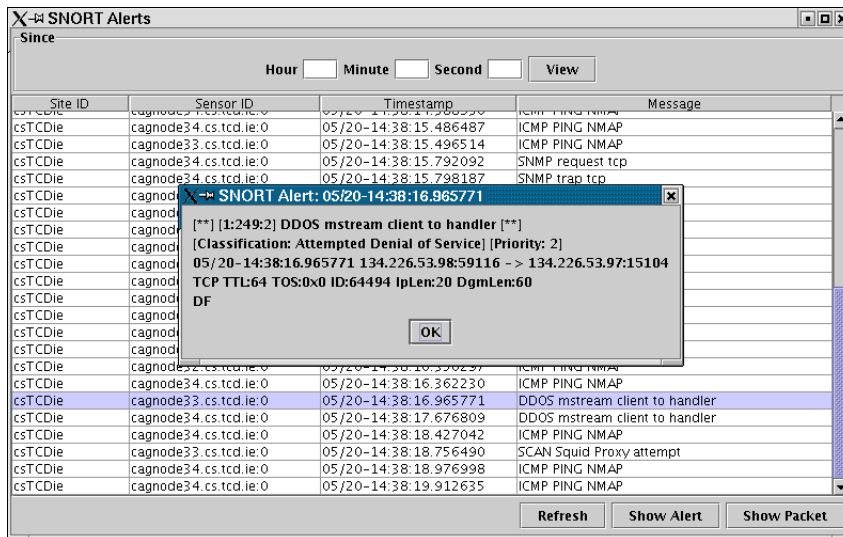


Figure 7.16: Viewer GUI, Snort alerts panel

# Chapter 8

## TESTING

Testing of the NetTracer system was divided into two stages. The first 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. The second phase was that of performance testing. This was to discover how the system performed in a proper test environment. 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 chapter describes these tests.

### 8.1 TEST DEPLOYMENT

The testbed cluster was installed using the LCFGng system, initially developed by the University of Edinburgh [12]. This cluster is comprised of seven nodes: four worker nodes (WN), a computing element (CE), a storage element (SE), and a user interface (UI). In addition to these a further node was needed to host and run the R-GMA Registry and Schema.

**Computing Element:** The computing element forms the entry point to the grid site. It is this node that receives jobs submitted to the grid, dispatches them for execution, and

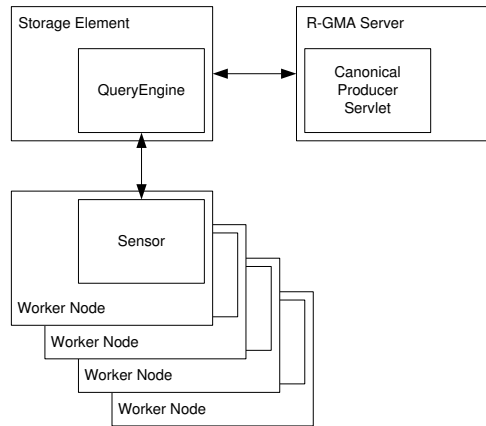
also returns the output.

**Worker Node:** These nodes are where the jobs submitted are actually run. The computing element dispatches received jobs to these nodes for execution. In TCD this is done using the Portable Batch System (PBS) software.

**Storage Element:** The storage element is used to hide the details of the backend storage systems. These could be disk arrays, or mass storage systems, etc. [12]

**User Interface:** This node provides the entry point to the grid for users. It provides the client software necessary to allow users to submit jobs to the grid, and to retrieve the output from these jobs.

For the test deployment a NetTracer Tcpdump sensor was installed on each worker node. The QueryEngine was installed on the storage element. As stated the R-GMA Registry and Schema services were installed on a separate node. The SE was also used to host the R-GMA producer servlets. Figure 8.1 summarises the test deployment.



**Figure 8.1:** Test deployment

## 8.2 FUNCTIONAL TESTS

These 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. This was caused by the Viewer 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 result 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 R-GMA and the QueryEngine to allow direct testing of the code.

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

## **8.3 PERFORMANCE TESTS**

These tests measure the performance of the NetTracer in the test environment. They are largely based on measuring the time taken to perform a defined set of queries. To perform these tests a Tcpdump logfile was acquired containing 100,000 packets (8Mb). The set of queries was performed on the file and the results recorded. The overall time, from query submission to query completion, as well as the time spent in each component, was recorded. The results of these tests are given in the following section.

## **8.4 RESULTS**

### **8.4.1 Time for ResultSet retrieval**

The first test measures the amount of time taken to return a set number of tuples (or packets) beginning at the start of the test file, in order to determine the performance of the system when retrieving result sets of various sizes. The times were recorded for a result set containing 1 tuple, 10 tuples, 100 tuples, 1,000 tuples, and 10,000 tuples. To execute the tests a simple consumer was written. This consumer submits a SQL query to the R-GMA and records the time that a result set is received in response. The query submitted was of the form:

```

SELECT * FROM Packet
WHERE sensorId = 'cagnode33.cs.tcd.ie:2'
AND fileId = 0
AND packetId < 1 | 10 | 100 | 1000 | 10000

```

Each test was executed 10 times and the average time taken was calculated. The following tables ( 8.1, 8.2, 8.3, 8.4, 8.5 ) show the results of these tests. Each table contains the total time the query took to complete, i.e. the time from submission of the query until the ResultSet was received, as well as the time taken by the QueryEngine to complete the query. The difference between the times is the time spent in the R-GMA system.

Run	Total Time (ms)	Time in QueryEngine (ms)
0	1299	37
1	1125	40
2	1482	38
3	1251	41
4	1085	37
5	1148	43
6	1077	37
7	1091	41
8	1093	53
9	1076	40

**Table 8.1:** Time for retrieval of 1 tuple

The last table, Table 8.6, shows the average times for the five sizes of ResultSet. Note that the standard deviation for the time spent in the QueryEngine for 100 tuples is only 42ms if one excludes run 0, perhaps indicating some exceptional condition for run 0.

Run	Total Time (ms)	Time in QueryEngine (ms)
0	1141	49
1	1123	43
2	1174	40
3	1167	46
4	1105	67
5	1148	40
6	1153	42
7	1116	40
8	1142	50
9	1151	61

**Table 8.2:** Time for retrieval of 10 tuples

Run	Total Time (ms)	Time in QueryEngine (ms)
0	1511	434
1	1178	165
2	1130	141
3	1144	104
4	1142	198
5	1332	82
6	1340	89
7	1132	84
8	1146	171
9	1191	85

**Table 8.3:** Time for retrieval of 100 tuples

Run	Total Time (ms)	Time in QueryEngine (ms)
0	5053	766
1	4068	755
2	2937	634
3	3219	601
4	3042	637
5	5063	685
6	4523	605
7	3779	579
8	3895	646
9	5998	680

**Table 8.4:** Time for retrieval of 1000 tuples

Run	Total Time (ms)	Time in QueryEngine (ms)
0	40139	8026
1	35251	7541
2	34341	7441
3	36598	7492
4	34868	7487
5	33691	7421
6	36978	7421
7	30801	7415
8	32936	7464
9	38231	7437

**Table 8.5:** Time for retrieval of 10000 tuples

Tuples	Total Time (mS)	Std. Dev.	Time in QueryEngine (ms)	Std. Dev
1	1172.7	126.4	40.7	4.5
10	1142	20.7	46.8	9.3
100	1224.6	121.3	155.3	101.2
1000	4157.7	947.3	658.8	59.8
10000	35383.4	2568.4	7514.5	184.1

**Table 8.6:** Average times for retrieval

## 8.4.2 Time for packet searching

The next test recorded the time taken to find specific sets of packets from different locations in the log file, the aim of the test being to determine the effect of log file size on the length of time the QueryEngine takes to process a query. Again the simple test consumer was used. Each test query was submitted 10 times and the average time taken recorded. The time spent in the QueryEngine was recorded, as well as the total time from query submission until the result set was received. The test file containing 100,000 tuples was again used. The query submitted searched for groups of packets based on timestamps. The first query selects a set of packets from the beginning of the file, the second from the middle of the file, and the third from the end of the file. The query used was of the form:

```
SELECT * FROM Packet
WHERE sensorId = 'cagnode33.cs.tcd.ie:2'
AND fileId = 0
AND timestamp_Secs > startTime
AND timestamp_Secs < (startTime + 10)
```

The results of these tests are shown in the tables below ( 8.7, 8.8, 8.9 ).

Run	Total Time (ms)	Time in QueryEngine (ms)
0	6623	3044
1	6357	3030
2	5825	3016
3	5570	2973
4	5745	2973
5	5218	2902
6	6804	2950
7	6224	2988
8	5349	2886
9	6027	2955

**Table 8.7:** Time for retrieval of tuples from beginning of file

Run	Total Time (ms)	Time in QueryEngine (ms)
0	5809	3457
1	5443	3447
2	5733	3409
3	6051	3441
4	5593	3504
5	6171	3362
6	5859	3436
7	6036	3452
8	5897	3340
9	6464	3431

**Table 8.8:** Time for retrieval of tuples from middle of file

Run	Total Time (ms)	Time in QueryEngine (ms)
0	5414	2844
1	6164	3094
2	5812	2944
3	6121	2852
4	5574	2923
5	5585	2858
6	5820	2952
7	5439	2845
8	5616	2969
9	5271	2956

**Table 8.9:** Time for retrieval of tuples from end of file

Table 8.10 shows the average times taken for ResultSet retrieval from each of the three file positions.

Position	Total Time (ms)	Std. Dev.	Time in QueryEngine (ms)	Std. Dev.
start	5974.2	502.2	2971.7	48.8
middle	5905.6	278.1	3427.9	45
end	5681.6	280.4	2923.7	78.4

**Table 8.10:** Average times for retrieval of tuples from file positions

### 8.4.3 Time per component

These tests record the time spent in each component during a query execution, in order to show where time is spent in the system. To perform these tests the Viewer was used. Times were taken from query submission by the Viewer until result set display in the Viewer query panel. Time taken was recorded for the Viewer, R-GMA, and QueryEngine, where:

- $T_{queryengine} = (\text{time query received by QueryEngine}) - (\text{time result set returned to R-GMA})$
- $T_{R-GMA} = (\text{time result set sent by QueryEngine}) - (\text{time result set received by Viewer})$
- $T_{viewer} = (\text{time result set received by Viewer}) - (\text{time result set displayed in result table})$

The following tables ( 8.11, 8.12, 8.13, 8.14, 8.15 ) show the results of these tests.

Component	Time (ms)
Viewer	140
R-GMA	1210
QueryEngine	49
<b>Total</b>	1399

**Table 8.11:** Times for retrieval of 1 tuple

Component	Time (ms)
Viewer	240
R-GMA	1034
QueryEngine	71
<b>Total</b>	1345

**Table 8.12:** Times for retrieval of 10 tuples

Component	Time (ms)
Viewer	297
R-GMA	1084
QueryEngine	291
<b>Total</b>	<b>1672</b>

**Table 8.13:** Times for retrieval of 100 tuples

Component	Time (ms)
Viewer	272
R-GMA	4757
QueryEngine	865
<b>Total</b>	<b>5894</b>

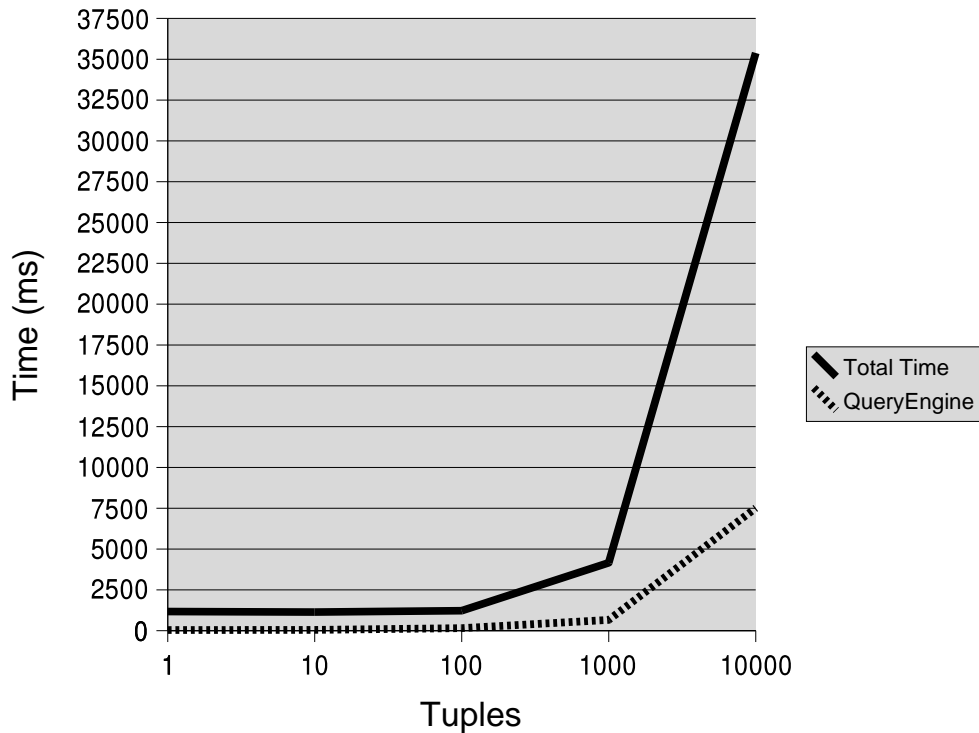
**Table 8.14:** Times for retrieval of 1000 tuples

Component	Time (ms)
Viewer	2757
R-GMA	28102
QueryEngine	8908
<b>Total</b>	<b>39767</b>

**Table 8.15:** Times for retrieval of 10000 tuples

## 8.5 RESULTS DISCUSSION

Figure 8.2 shows a graph of the average times for the retrieval of the different sizes of result set, from 1 tuple to 10,000 tuples.

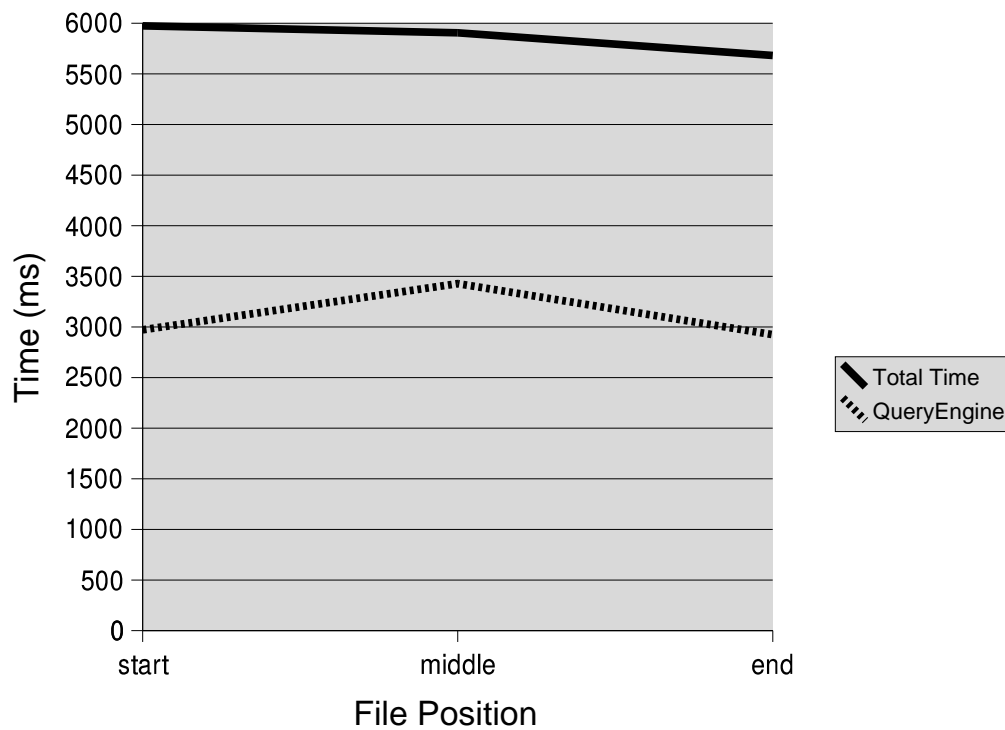


**Figure 8.2:** Average times for retrieval

The graph clearly shows that the most important factor in determining the length of time a query will take to execute is the size of the returned result set. Initially the times are very similar, for 1 to 100 tuples. For result sets above this the query time increases rapidly. The results also clearly show that the other major factor is the *performance of the R-GMA*. The average total query time for a result set containing 10,000 tuples was 35,383.4ms, although the QueryEngine in fact completed the query in 7,514.5ms. The bulk of the remaining time

is spent in the R-GMA Consumer servlet, see later.

The second test was an attempt to determine the impact of log file size on query processing times. There have been no attempts thus far at query processing optimisation; the QueryEngine performs a simple linear search of the log file, checking each packet in the file to see if it satisfies the query. It would be expected, therefore, that the larger the log file, the longer the query would take to complete. To show this queries were submitted that selected tuples from different positions in the log file, the start of the file, the middle, and the end of the file. Figure 8.3 shows a graph of the average times for tuple retrieval from the different file positions.

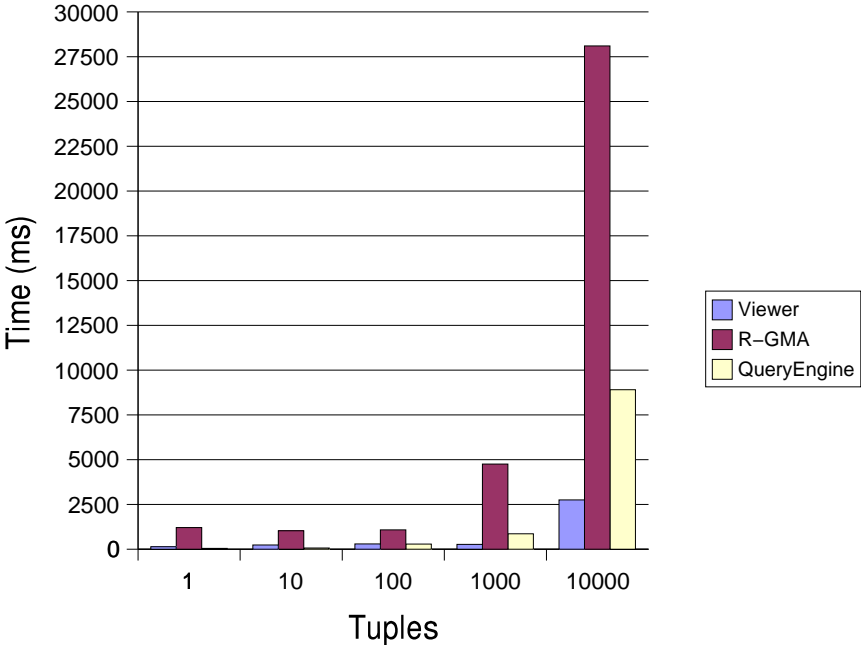


**Figure 8.3:** Average times for retrieval of tuples from file positions

The graph, however, shows that this is not in fact the case. There is no significant increase

in the amount of time taken between sets of packets selected from the start of the file and those from the end. The slight peak in the graph for the set of packets selected from the middle of the file is caused by a having to use a slightly more complex query in order to keep the sizes of the result sets consistent. Therefore it can be said that log file size has a far lower impact on performance than could be expected. Despite the lack of optimization of the search algorithm, the performance of the simple linear search seems to scale well with increases in log file size.

As stated the other major factor is the performance of the R-GMA system itself. The next set of tests are designed to show this quite clearly, by measuring the amount of time spent in each component during query processing, where the times per component are measured as defined in Section 8.4.3. Figure 8.4 shows a comparison of the times spent in each component during query execution.



**Figure 8.4:** Time spent in each component during query execution

This clearly shows the importance of the R-GMA performance on the overall time. For example, in the longest case of 10,000 tuples, 70% of the overall time was spent in transferring the result set from the producer servlet to the consumer. Therefore only 10% is spent in the Viewer and 20% in the QueryEngine (this component is clearly able to be substantially optimized, see Chapter 12). The time spent in the Viewer, from query reception to results display, is caused by having to cycle through the result set and add the data to a Java table model for display. Obviously the larger a result set the longer this process takes.

The time spent in the R-GMA is largely due to the Consumer servlet, since the Canonical-Producer servlet simply acts as an intermediary between the QueryEngine and the Consumer servlet. As soon as the CanonicalProducer servlet begins receiving the XML format result set from the QueryEngine, the result set is forwarded directly to the Consumer servlet. When the Consumer servlet receives the result set, in XML format, it is converted to an R-GMA ResultSet object and stored in the Consumer servlet's queue. In order to return the result set to the Consumer object, the servlet must extract the result set from its queue and convert it back to XML for transmission to the Consumer. The Consumer receives this XML result set and then performs a further conversion from XML to a ResultSet object. There is therefore a total of three conversions between XML and ResultSet objects. Each of these conversions is quite expensive, particularly when dealing with large result sets. This is clearly an area where some optimization would benefit R-GMA.

## Chapter 9

# EXAMPLE EXPERIMENTS

The purpose of a SANTA-G system is to allow for ad-hoc monitoring experiments in the grid environment. 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 example experiments that utilise the NetTracer.

### 9.1 TCP THROUGHPUT MEASUREMENTS

The first example experiment is to obtain throughput measurements using the network monitoring data obtained from the 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 Tcpdump 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.

### 9.1.1 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 packet capture 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 under test. 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.

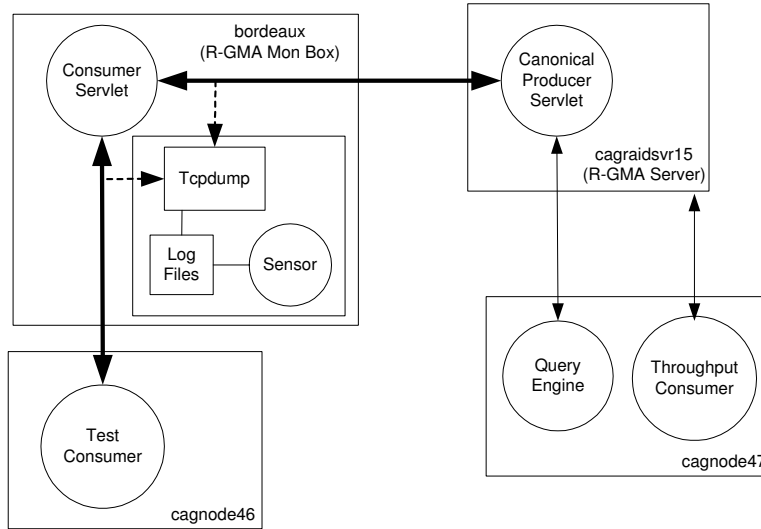
The nodes involved in the experiment are as follows (see Figure 9.1):

**cagnode46.cs.tcd.ie:** hosts the test consumer.

**bordeaux.cs.tcd.ie:** hosts the Consumer Servlet, and the Tcpdump 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.



**Figure 9.1:** Throughput measurements deployment

The Tcpdump 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 (or port 8443 if using HTTPS). 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.

### 9.1.2 Write the Consumer code

In order to access the data acquired by the NetTracer 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 NetTracer 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.

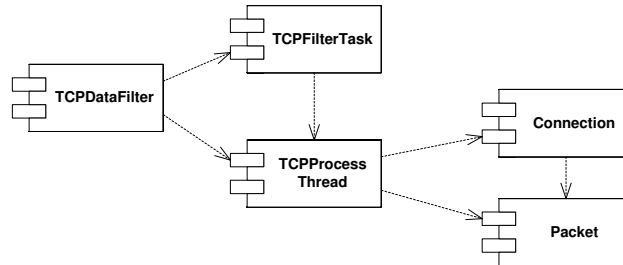
When the Consumer completes, the result set can be obtained by calling one of the Consumer's `pop()` methods. `popIfPossible()` will return a result set if available, or null if no result set was returned.

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).

The throughput consumer is composed of five separate classes (see Figure 9.2): `TCPDataFilter`, `TCPFilterTask`, `TCPProcessThread`, `Connection` and `Packet`. `TCPDataFilter` is the main class, used to instantiate the `DatabaseProducer` that both stores and publishes the results of the calculations, and also creates and starts a Java Timer object and the processing thread. The Timer object runs the `TCPFilterTask` thread at regular intervals. The `TCPFilterTask` thread uses a Consumer object to obtain the required subset of the data needed to perform the throughput calculations. The `ResultSets` obtained by the `TCPFilterTask` are then passed to the `TCPProcessThread`, which calculates the TCP throughput values and inserts them into the `DatabaseProducer`.

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,



**Figure 9.2:** TCP throughput Consumer components

the TCPFilterTask 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:

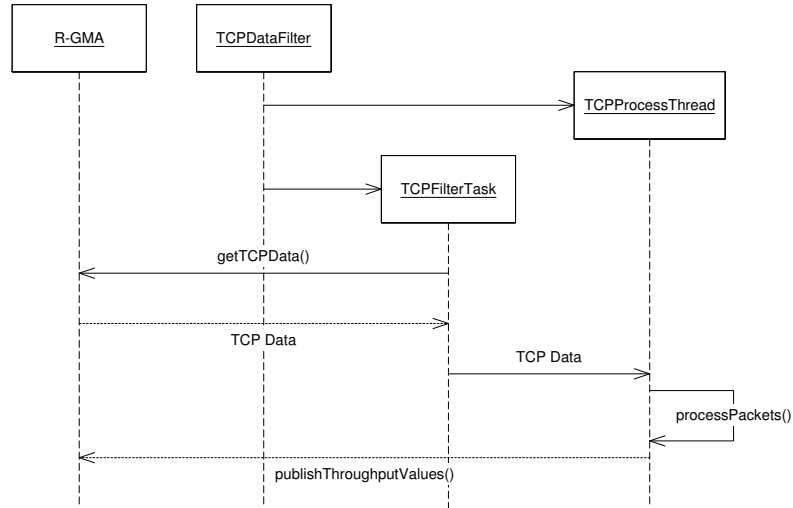
```

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 the TCPProcessThread. Figure 9.3 shows a sequence diagram for this.

When the process thread receives the ResultSet, each row is used to construct a Packet object. The packet object encapsulates all the data needed from the TCP packet in order to calculate the throughput values. Each packet object is checked against the set of currently open Connection objects to see if it belongs to that connection. Each TCP connection is uniquely identified by the source host and port and destination host and port. These values



**Figure 9.3:** Throughput Consumer sequence diagram

are used to match a packet to a connection. If a packet does belong to an existing connection it is added to it, and the connection’s current throughput values are updated. If it does not, the packet is checked to see if it is a SYN packet (the TCP packet used to open a new connection), and if so, a new Connection object is created to represent this TCP connection. The new Connection object is then added to the list of currently open connections. Each time a packet is added to a connection the average throughput seen so far is calculated and a row published in the AverageThroughput table. When a connection is closed the final throughput values are calculated and a row published in the TCPThroughput table. The throughput is calculated and published for both directions in the connection, i.e. host A to host B and host B to host A.

HostA	HostB	Throughput	
		AtoB	BtoA
cagnode33.cs.tcd.ie:35556	cagraidsvr09.cs.tcd.ie:80	19026	24022
cagnode33.cs.tcd.ie:35560	cagraidsvr09.cs.tcd.ie:80	19450	24557
cagnode33.cs.tcd.ie:35565	cagnode34.cs.tcd.ie:22	277	341

**Table 9.1:** Sample throughput values obtained

The output from the Consumer has been 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. Table 9.1 shows a sample of results published dynamically during testing of the throughput Consumer whilst acquiring packet data with the NetTracer from a single test host (in this case cagnode33.cs.tcd.ie).

The following shows the results obtained when the log file acquired on the test host was analysed with the Tcptrace application.

```
TCP connection 2:
host c:      cagnode33.cs.tcd.ie:35556
host d:      cagraidsvr09.cs.tcd.ie:80
complete conn: yes
first packet: Tue Apr  6 11:00:46.118738 2004
last packet:  Tue Apr  6 11:00:46.128146 2004
elapsed time: 0:00:00.009408
total packets: 10
filename:    /opt/santag/examples/traces/tpptest/cagnode33_1.log
  c->d:      d->c:
    total packets:      5          total packets:      5
    throughput:         19026 Bps  throughput:         24022 Bps
```

```
TCP connection 8:
host o:      cagnode33.cs.tcd.ie:35560
host p:      cagraidsvr09.cs.tcd.ie:80
complete conn: yes
first packet: Tue Apr  6 11:06:47.185818 2004
last packet:  Tue Apr  6 11:06:47.195021 2004
elapsed time: 0:00:00.009203
total packets: 10
filename:    /opt/santag/examples/traces/tpptest/cagnode33_1.log
```

```

o->p:          p->o:
  total packets:          5          total packets:          5
  throughput:            19450 Bps   throughput:            24557 Bps

```

TCP connection 13:

```

host y:          cagnode33.cs.tcd.ie:35565
host z:          cagnode34.cs.tcd.ie:22
complete conn:  yes
first packet:    Tue Apr  6 11:13:48.705799 2004
last packet:     Tue Apr  6 11:13:56.792011 2004
elapsed time:    0:00:08.086211
total packets:   72
filename:        /opt/santag/examples/traces/tptest/cagnode33_1.log
y->z:           z->y:
  total packets:          42          total packets:          30
  throughput:            277 Bps     throughput:            341 Bps

```

As can be seen the Tcptrace output agrees with that obtained by the throughput Consumer. Then why use the NetTracer rather than Tcptrace? Because the NetTracer enables multiple real-time acquisitions at geographically dispersed sites, and consequent calculations to be performed in a grid-enabled fashion, in contrast to a single off-line local equivalent.

### 9.1.3 Run the experiment

The final step is to run the experiment. To do this a test sensor must be started on a node in the testbed. The *Tcpdump* 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 test sensor's logfile, should then be started.

The table below (Table 9.2) 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 9.2:** 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, used to submit a query to the R-GMA system, see Table 9.3. The code used is the same as that outlined in Section 9.1.2. 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 9.2 that the connection is not closed until after the call to close the Consumer API object. The 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 9.4 shows the first measurement for the connection. This corresponds to the transmission of the SQL query.

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 9.3:** 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 9.4:** SQL query transmission measurement

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 **1381.76ms**. It is known, from the QueryEngine logs that it took **938ms** for the QueryEngine to complete the query. This implies the CanonicalProducer Servlet, not taking into account network delays, added an additional **443.76ms**.

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 **1690.44ms**, which corresponds to approximately 55% of the total **3072ms** (as calculated from the connection times) taken to answer the query. Table 9.5 summarises the

times observed:

Measurement	Time
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
implied time for XML to ResultSet conversion	427
total query time as measured by test Consumer	3449

**Table 9.5:** Summary of calculated times (ms)

The additional time measured by the test Consumer (the difference between 3449ms and 3072ms, i.e. 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.

The implication is that the majority of the time is taken up by transmission of the ResultSet, and that despite its simple linear search algorithm the QueryEngine does not greatly degrade performance. It also justifies the premise of the CanonicalProducer and SANTA-G, to query information in place, and thereby not move bulk raw data across the network.

## 9.2 MPI RING MEASUREMENTS

In this experiment the NetTracer is used to trace MPI packets generated by an application running across several sites in a grid. The application used is a simple MPI ring program that sends packets of a configurable length around a ring of MPI processes. Each process will run in a separate Grid-Ireland site, communicating with each other directly using Globus I/O. It is these inter-site communications that we wish to collect using NetTracer sensors. The purpose is to calculate the latency (i.e. the total time taken to travel around the MPI ring) and effective bandwidth (i.e. the total time divided by the total bytes sent in that time) for the traced packets.

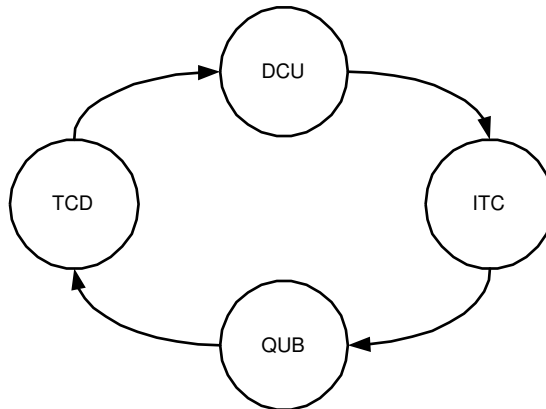
Although the output from the MPI ring program does include the total time and the effective bandwidth for each packet sent by the application, these measurements include the time spent processing the packet at each site, i.e. the time between a process receiving a packet and the time that it sends the packet to the next process in the ring. This time could be referred to as the ‘host overhead’. By analysing the traffic collected at each site by the Tcpdump sensors it is possible to calculate this overhead, and to therefore exclude these from the time and bandwidth calculations so that the measurements obtained are based purely on time spent on the network links. Ranking sites based on the host overhead experienced also provides a means of selecting the optimum set of sites for an application run.

Again the three steps described above must be carried out in order to run the experiment.

### 9.2.1 Configure the SANTA-G system

For the experiment 4 Grid-Ireland sites are chosen, TCD (Trinity College Dublin), DCU (Dublin City University), ITC (Institute of Technology, Carlow), and QUB (Queens University Belfast). Once the MPI application is submitted, an MPI process will be started on a single node at each of the selected sites.

As stated the processes communicate using Globus I/O. Globus I/O uses a range of ports



**Figure 9.4:** Direction of MPI ring

defined in a site's configuration, usually 20000 to 25000. The application source code has also been modified so that the packets generated contain the word 'RING' within the data payload. This information can be used to identify the traffic generated by the application, therefore, the appropriate Tcpcdump arguments used in the configuration of the sensors are:

```

((tcp[0:2] >= 20000 and tcp[0:2]<= 25000) and
(tcp[2:2] >= 20000 and tcp[2:2] <= 25000)) and
(tcp[32:4] = 0x52494e47)

```

These arguments tell Tcpcdump to collect all packets with a source and destination port between 20000 and 25000 (the Globus port range), and that contain the value '0x52494e47' (the hex ASCII value of the word 'RING') in the data payload.

### 9.2.2 Write the Consumer code

We wish to calculate the total time taken by a packet to traverse the ring, not including the time spent processing the packet on a node. To do this the consumer must query for all packets collected by each of the sensors. The packet timestamps must be extracted from the returned result sets and converted from seconds and microseconds fields to a single microsecond value.

The time spent processing a packet at a site,  $\Delta T_{p_i}$ , where  $i$  is the MPI process number from 0 to 3, is calculated as the time between a packet being received by a process,  $T_r$ , and the time the packet is sent to the next process in the ring,  $T_s$ :

$$\Delta T_{p_i} = T_s - T_r$$

The total time taken is given by  $\Delta T_{p_0}$ , as any packet received by process 0 will have traversed the entire ring of MPI processes. By using relative times we may ignore clock offsets between sites. The time without host overheads can therefore be calculated as follows:

$$\Delta T_{nettracer} = \Delta T_{p_0} - \sum_{i=1}^3 T_{p_i}$$

The effective bandwidth is then given by:

$$\Delta Bw_{nettracer} = \frac{(packetLength * noOfProcesses)}{\Delta T_{nettracer}}$$

The latency and effective bandwidth calculated from the application traffic by the consumer is then compared to the output of the MPI application. Plots of the bandwidth and latency distributions are generated by the consumer, as well as distributions of the differences between the two sets of results (i.e. those obtained from the MPI application and those from the consumer).

### 9.2.3 Run the experiment

To run the experiment first a sensor, configured as described above, must be started on each node that will host an MPI process. The MPI application is submitted to the 4 chosen sites using a Globus RSL file of the following form:

```
+
( &(resourceManagerContact="gridgate.cs.tcd.ie:2119/jobmanager-lcgpbs")
  (queue=test)
  (count=1)
  (label="subjob 0")
  (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
    (LD_LIBRARY_PATH /opt/globus/lib/))
  (executable=$(GLOBUSRUN_GASS_URL) # "mpitest")
  (arguments=1000 1000)
```

```

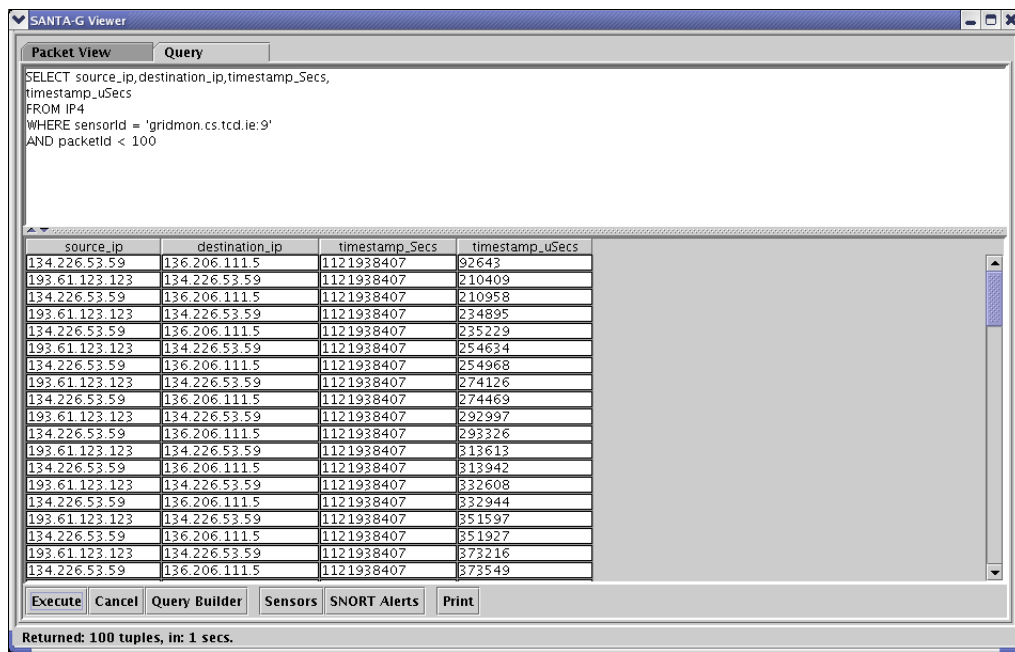
        (stdout=$(GLOBUSRUN_GASS_URL) # mpioutput.0)
    )
( &(resourceManagerContact="gridgate.besc.ac.uk:2119/jobmanager-lcgpbs")
    (queue=test)
    (count=1)
    (label="subjob 1")
    (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1)
        (LD_LIBRARY_PATH /opt/globus/lib/))
    (executable=$(GLOBUSRUN_GASS_URL) # "mpitest")
    (arguments=1000 1000)
    (stdout=$(GLOBUSRUN_GASS_URL) # mpioutput.1)
)
( &(resourceManagerContact="gridgate.itcarlow.ie:2119/jobmanager-lcgpbs")
    (queue=test)
    (count=1)
    (label="subjob 2")
    (environment=(GLOBUS_DUROC_SUBJOB_INDEX 2)
        (LD_LIBRARY_PATH /opt/globus/lib/))
    (executable=$(GLOBUSRUN_GASS_URL) # "mpitest")
    (arguments=1000 1000)
    (stdout=$(GLOBUSRUN_GASS_URL) # mpioutput.2)
)
( &(resourceManagerContact="gridgate.dcu.ie:2119/jobmanager-lcgpbs")
    (queue=test)
    (count=1)
    (label="subjob 3")
    (environment=(GLOBUS_DUROC_SUBJOB_INDEX 3)
        (LD_LIBRARY_PATH /opt/globus/lib/))
    (executable=$(GLOBUSRUN_GASS_URL) # "mpitest")
    (arguments=1000 1000)
    (stdout=$(GLOBUSRUN_GASS_URL) # mpioutput.3)
)

```

The application will send 1,000 packets of 1,000 bytes in size around the ring of processes. This is specified in the arguments field of the RSL file, (arguments=1000 1000). An MPI process starts on a single node at each of the sites to which the application was submitted:

21/07/2005 10:33:26 Process 1 is alive on gridmon.besc.ac.uk  
 21/07/2005 10:33:26 Process 2 is alive on gridmon.itcarlow.ie  
 21/07/2005 10:33:26 Process 3 is alive on gridmon.dcu.ie  
 21/07/2005 10:33:27 Process 0 is alive on gridmon.cs.tcd.ie

Once the application completes it is possible to query for the packets collected during the application run using the Viewer GUI (see Figure 9.5).



**Figure 9.5:** Captured MPI packets in Viewer GUI

Table 9.6 shows the data obtained for a single packet traversing the ring of MPI processes. The data payload of the packet is published by the NetTracer in the data table. By querying this table we can see the 'RING' tag that was introduced into the packet payload in order to allow for the packets sent by the application to be identified and traced.

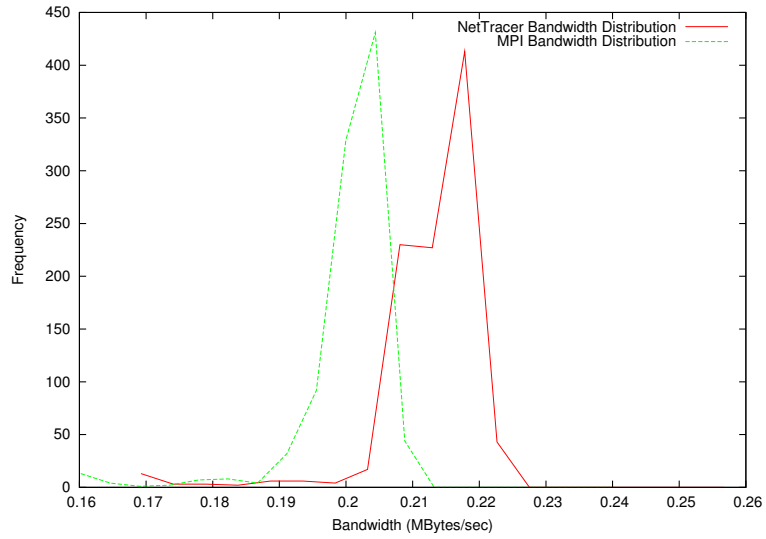
Figure 9.7(a) shows the latency as measured by the MPI application, and that calculated by the consumer from the raw network traffic. The effective bandwidth calculated from the

Src	Dst	Src Time	Dst Time	Data
gridmon.cs.tcd.ie	gridmon.dcu.ie	10:33:27.235229	10:33:27.236349	...fc675b352494e470016...
gridmon.dcu.ie	gridmon.itcarlow.ie	10:33:27.236671	10:33:27.240061	...fc675b352494e470016...
gridmon.itcarlow.ie	gridmon.besc.ac.uk	10:33:27.240411	10:33:27.247640	...fc675b352494e470016...
gridmon.besc.ac.uk	gridmon.cs.tcd.ie	10:33:27.247853	10:33:27.254634	...fc675b352494e470016...

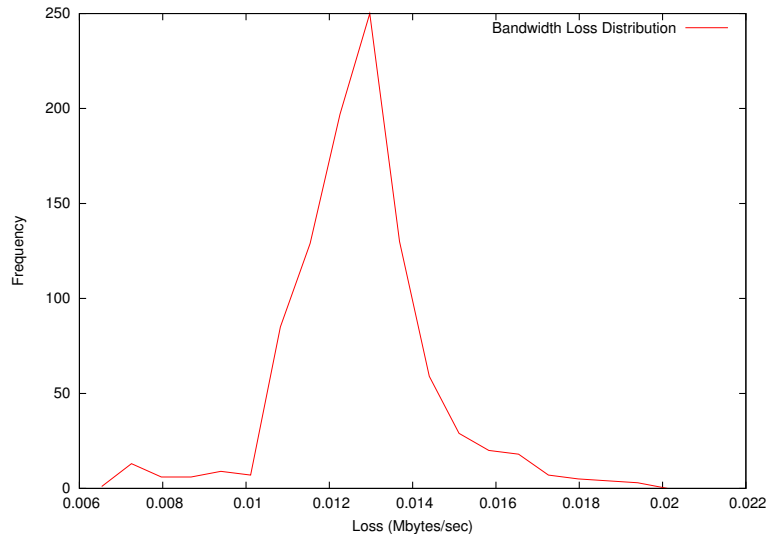
**Table 9.6:** Data collected by sensors for single ring transmission

measured latencies is shown in Figure 9.6(a). The distribution of the loss (i.e. the difference between the values including the host overheads and those without) is shown in Figures 9.6(b) and 9.7(b). On average there is approximately a 13 kB/sec bandwidth decrease ( $\sim 6\%$ ) and a  $1200\mu s$  increase in latency ( $\sim 6.5\%$ ) due to host overheads such as TCP/IP stack processing, Xen virtualisation (the nodes upon which the MPI application was executed are virtual machines), etc.

The effects of geographical distance between sites are very clear. The short hop from TCD to DCU takes little time, while the long hops to Carlow and on to QUB and back to TCD take correspondingly, more time. The effects of competition for bandwidth utilization are less clear. The NREN backbone links are overprovisioned for QoS, and mostly the network paths are 1 Gbps (but see the next experiment) through routers, switches, and firewalls. The low effective bandwidth (approximately 0.21 MBytes/sec) should be a reality check to those wishing to execute message passing programs across geographically dispersed sites, whatever the performance of the network and site resources.

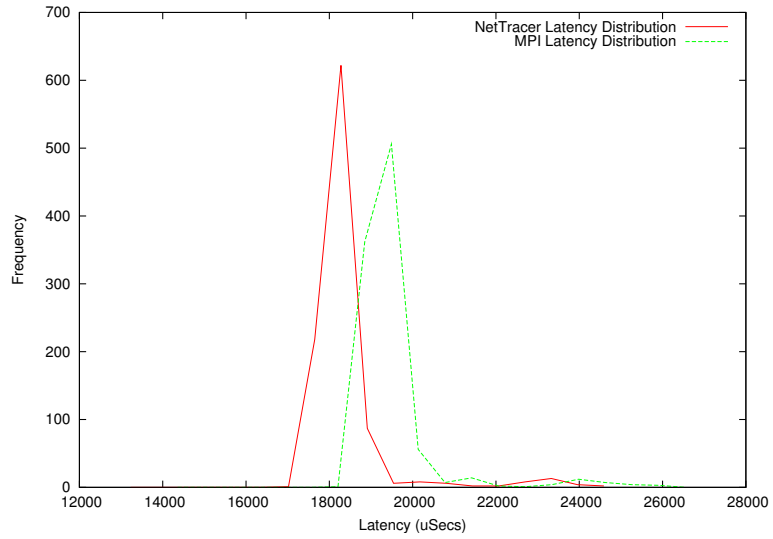


(a) Effective bandwidth

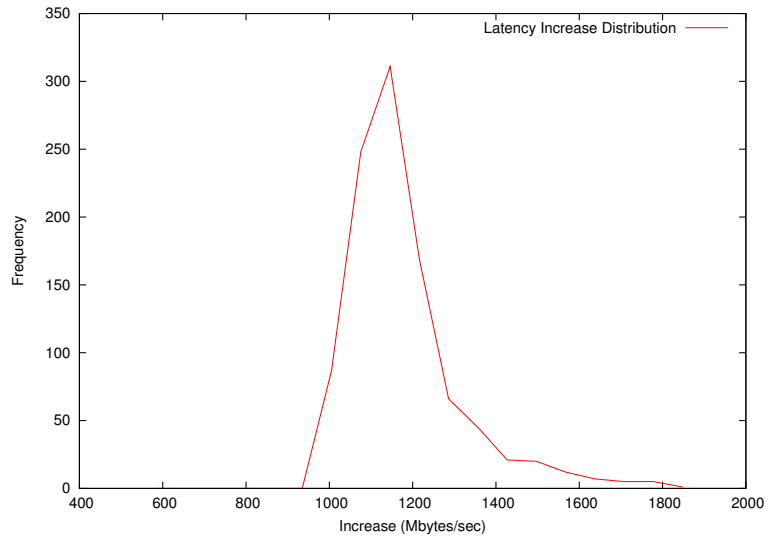


(b) Effective bandwidth loss

**Figure 9.6:** MPI Ring and NetTracer effective bandwidth distributions



(a) Latency



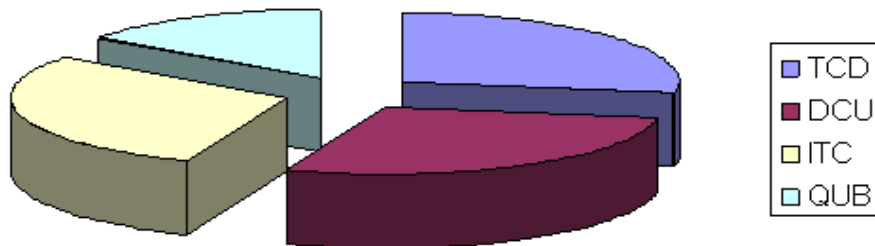
(b) Latency increase

**Figure 9.7:** MPI Ring and NetTracer latency distributions

It is possible to calculate the average time spent processing packets at each site by averaging the overhead experienced by each packet as it traverses the ring, see Table 9.7. As can be seen QUB performs significantly better than either TCD, DCU or ITC. Using this approach it would be possible to optimise the performance of an application by choosing sites with the lowest overhead when submitting the application to the grid.

Site	Overhead ( $\mu s$ )
TCD	334.97
DCU	343.76
ITC	353.77
QUB	183.62

**Table 9.7:** Average overhead per site



**Figure 9.8:** Overhead per site

## 9.3 ONE-WAY LATENCY MEASUREMENTS

The aim of this experiment is to measure the one-way latency for packets sent between two Grid-Ireland sites in a similar fashion as is done by RIPE equipment [41], but at a finer grain, and without the associated fees, and also with unrestricted access to the raw data. The MPI application described in the previous experiment is again used in order to generate traffic for the measurements. An MPI ring is created with only two processes, sending packets back and forth for a specified number of iterations. By using NetTracer Tcpdump sensors this traffic can be traced, and the latencies experienced by packets between sites can be calculated by subtracting the send time of a packet from the receive time. Remember that for the NetTracer Tcpdump is unmodified, so its behaviour is well understood.

### 9.3.1 Configure the SANTA-G system

For this experiment the TCD and QUB Grid-Ireland sites were chosen. Each site must host a NetTracer Tcpdump sensor. A similar configuration of the NetTracer as was used in the previous experiment is again required. Packets sent by the MPI application will again be within the Globus port range of 20000 to 25000. Here the source code has been modified so that the MPI packets contain the tag ‘SANTAG’ (0x53414e544147 in hex), therefore, the Tcpdump arguments needed are the following:

```
((tcp[0:2] >= 20000 and tcp[0:2]<= 25000) and
(tcp[2:2] >= 20000 and tcp[2:2] <= 25000)) and
(tcp[32:4] = 0x53414e54 and tcp[36:2] = 0x4147)
```

### 9.3.2 Write the Consumer code

The time taken for a packet to travel from site A to site B,  $\Delta T_{fwd}$ , can be expressed as follows, where  $T_{owl}$  is the one-way latency, and  $T_{offset}$  is the offset between the host clock at

site A and the host clock at site B:

$$\Delta T_{fwd} = T_{owl} + T_{offset}$$

The one-way latency in the reverse direction, i.e. site B to site A,  $\Delta T_{rev}$ , is therefore given by:

$$\Delta T_{rev} = T_{owl} - T_{offset}$$

Hence, the average one-way latency and offset for the forward and reverse journey can be calculated as:

$$T_{owl} = \frac{(T_{fwd} + T_{rev})}{2}, T_{offset} = \frac{(T_{fwd} - T_{rev})}{2}$$

The consumer uses these equations to calculate the average one-way latencies for packets sent between the two sites. Packets collected by the sensors are matched by source, destination, and datagram ID, and their timestamps subtracted. The consumer generates plots of the average one-way latencies, as well as the calculated offsets. Plots of the distribution of these values are also generated.

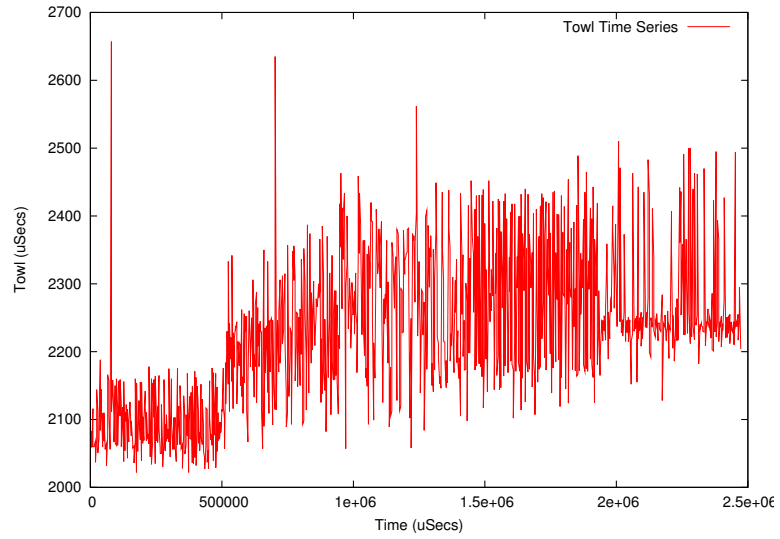
### 9.3.3 Run the experiment

We wish to determine how the one-way latency varies between the TCD and QUB sites over time. To do this measurements must be made at regular intervals. A dataset is collected once every hour. Each dataset represents a single run of the MPI application. The application sends 1,000 packets of 100 bytes in size during each execution (the packet size is as for RIPE). Once the application completes, the consumer queries the R-GMA for the packets generated by the application and calculates the average one-way latencies as described above. For each dataset the consumer generates the following output, which is stored for later analysis:

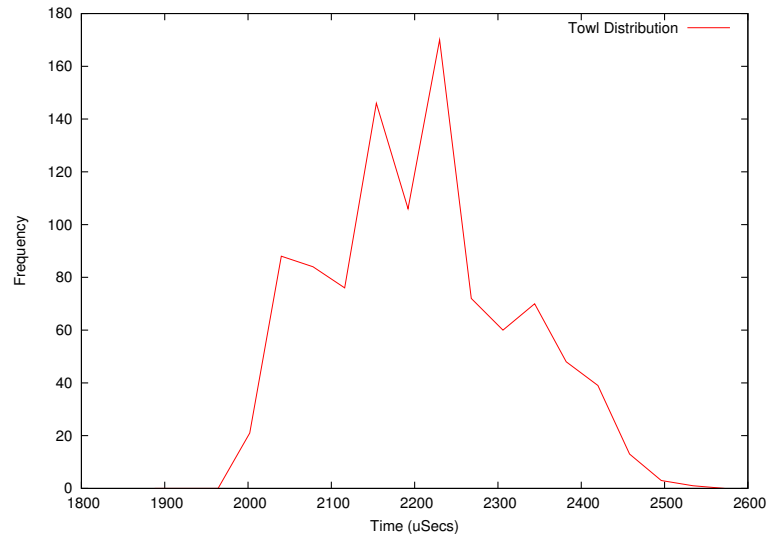
- Timestamps of all packets generated
- Minimum and maximum  $T_{owl}$  and  $T_{offset}$
- Mean and standard deviation of  $T_{owl}$  and  $T_{offset}$

- $T_{owl}$  time series and distribution plots
- $T_{offset}$  time series and distribution plots

Figure 9.9(a) shows the values of  $T_{owl}$  obtained for a typical dataset from a single application execution using the TCD and QUB sites. Figure 9.10(a) shows the values of  $T_{offset}$  calculated for the same dataset. Figures 9.9(b) and 9.10(b) show the distribution of the values obtained. The consequent effects of competition for bandwidth are visible, as is the consequent discretisation resulting from traversing alternate routes.

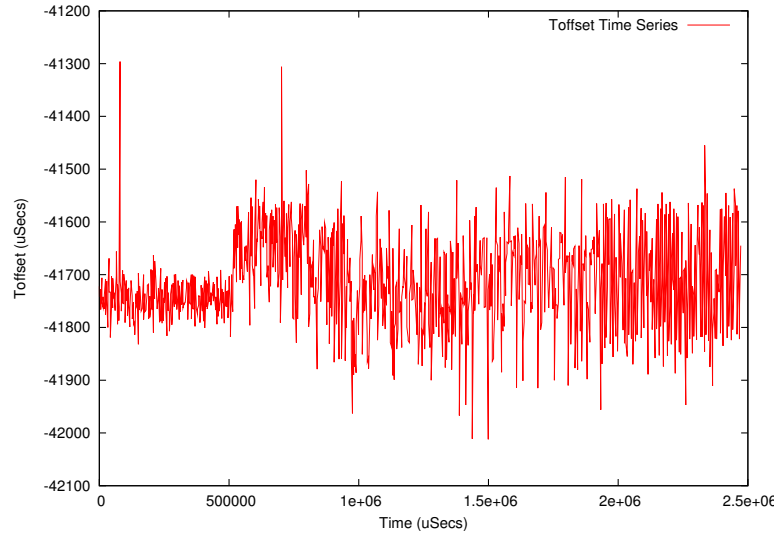


(a) Time series

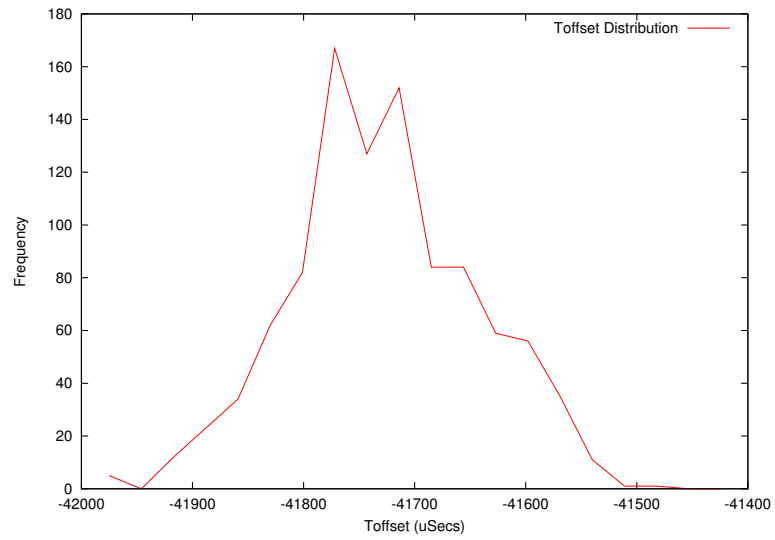


(b) Distribution

**Figure 9.9:**  $T_{owl}$  measurements for single application execution



(a) Time series



(b) Distribution

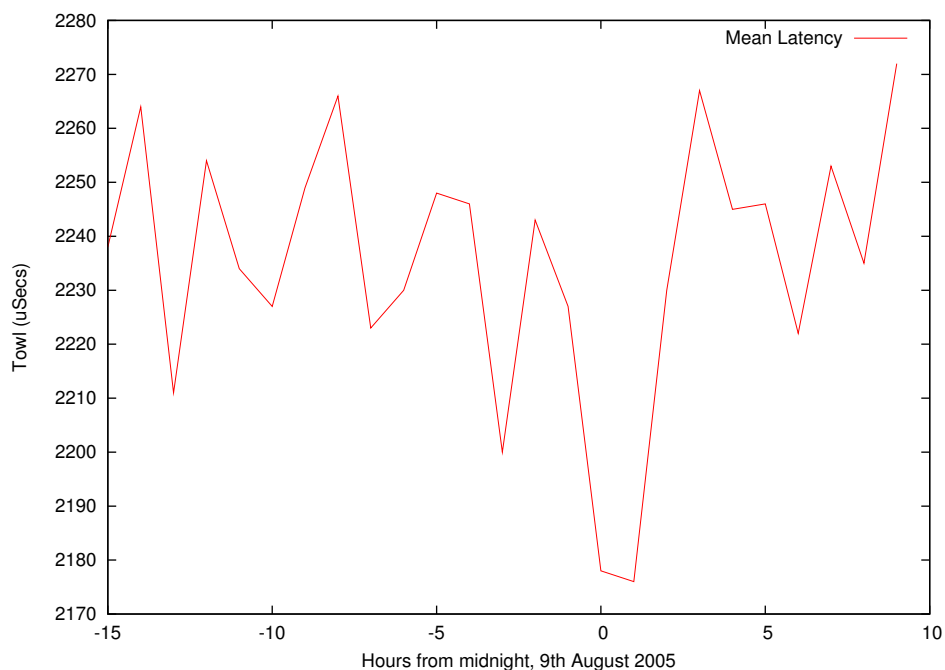
**Figure 9.10:**  $T_{offset}$  measurements for single application execution

Series	Start	End
1	2005-08-09 09:00	2005-08-10 09:00
2	2005-08-10 09:00	2005-08-11 09:00
3	2005-08-11 09:00	2005-08-12 09:00

**Table 9.8:** Collected dataset series

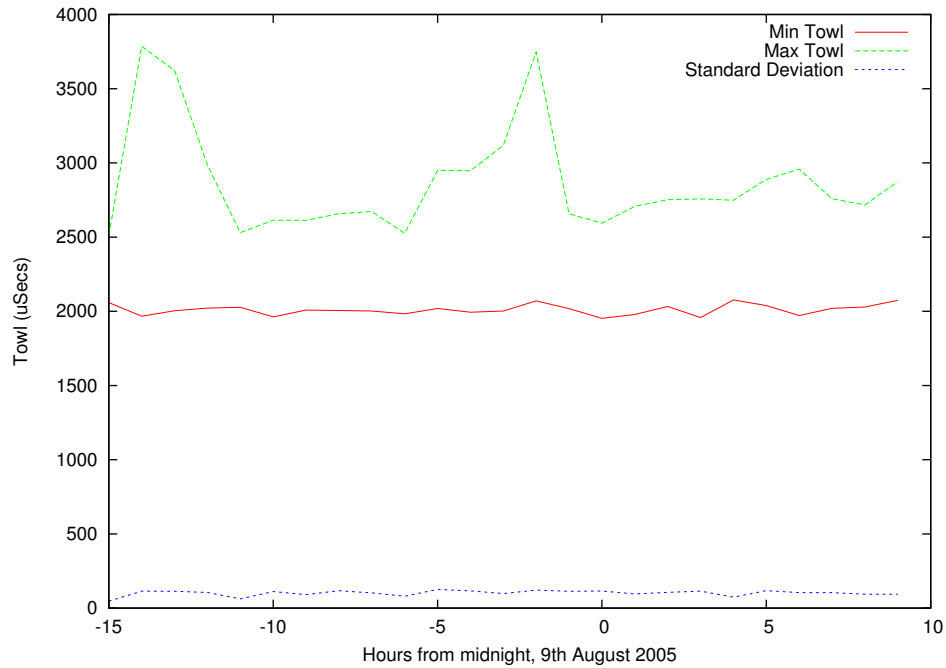
Datasets were collected over a continuous 72 hour period, from 09:00 on the 9th of August, until 09:00 on the 12th of August 2005. The datasets were divided into separate series, each corresponding to consecutive 24 hour periods, as shown in Table 9.8.

Figure 9.11 plots the mean  $T_{owl}$  values for the first day (series 1). Figure 9.12 plots the minimum, maximum and standard deviation of  $T_{owl}$  for series 1.



**Figure 9.11:** Mean  $T_{owl}$  values, series 1

The mean latency over the first 24 hour period was  $2235\mu s$ , with a standard deviation of  $25\mu s$ . The minimum latency values were experienced around midnight, with the minimum

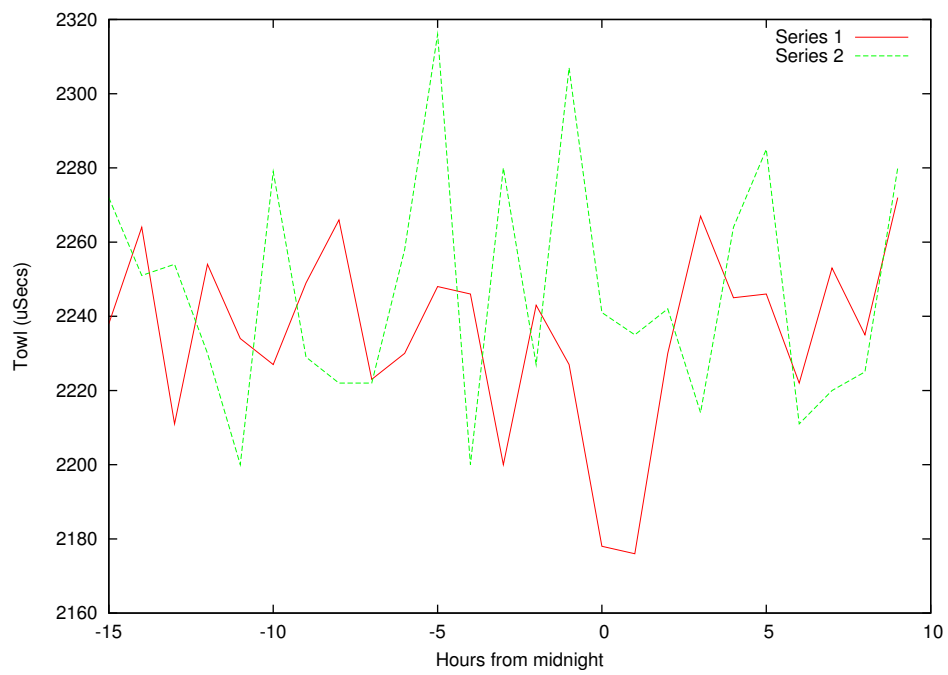


**Figure 9.12:**  $T_{owl}$  minimum, maximum and standard deviation, series 1

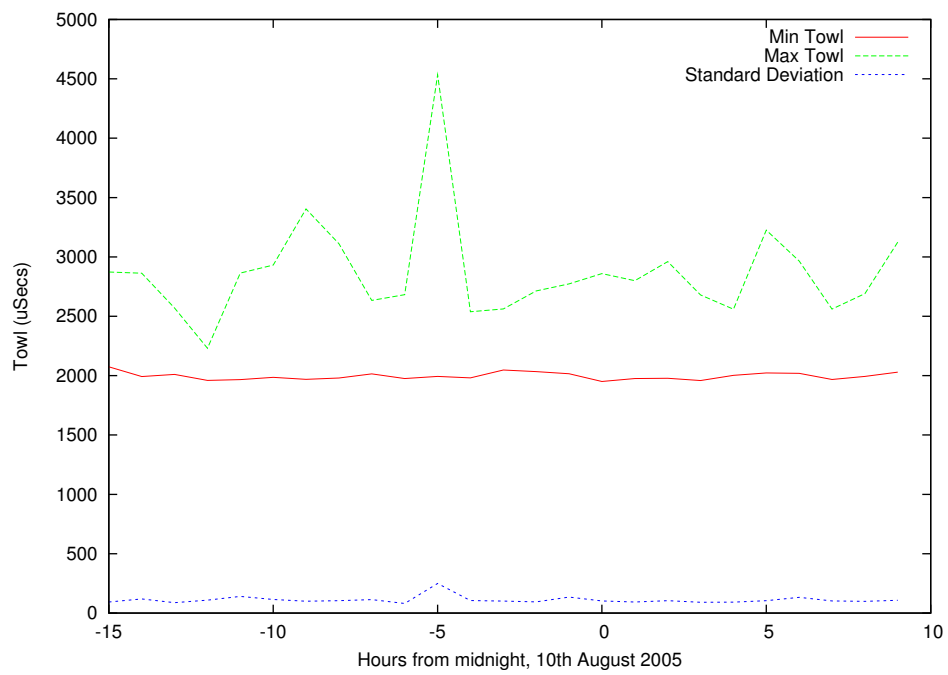
mean value of  $2176\mu s$  occurring at 01:00 on the 10th of August. The maximum value of  $2272\mu s$  was seen at 09:00 on the 10th of August.

Figure 9.13 plots the mean latency values of both series 1 and series 2. The minimum, maximum, and standard deviation of series 2 are shown in Figure 9.14.

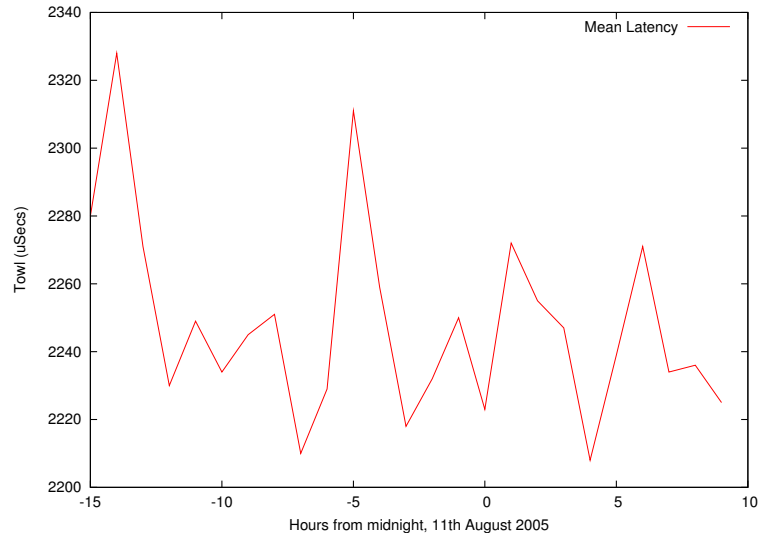
For the second day (series 2) the mean latency was  $2247\mu s$ , with a standard deviation of  $32\mu s$ . There was no repeat of the reduction in latency values experienced at midnight in series 1. The minimum mean value for series 2, of  $2200\mu s$ , was experienced at 20:00 on the 10th of August. For the third day the mean latency was  $2248\mu s$ , with a standard deviation of  $29\mu s$  (see Figures 9.15(a) and 9.15(b)). Figure 9.16 shows mean latencies for the full 72 hour measurement period.



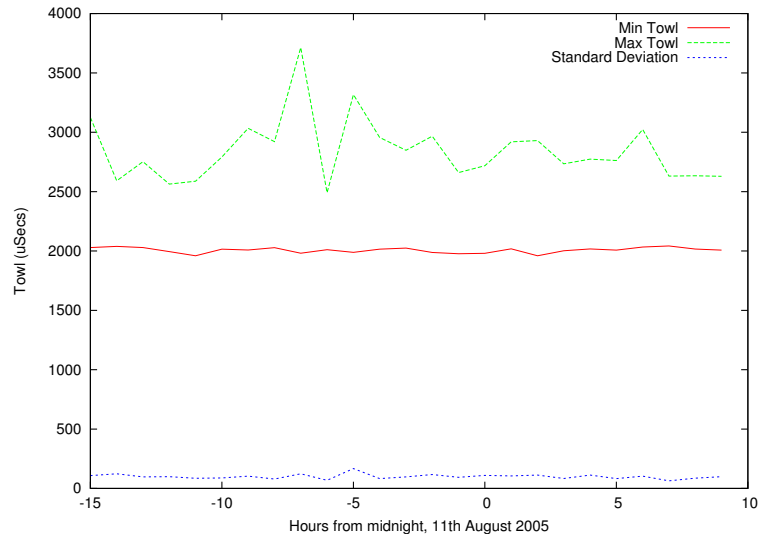
**Figure 9.13:** Mean  $T_{owl}$  values, series 1 and 2



**Figure 9.14:**  $T_{owl}$  minimum, maximum, and standard deviation, series 2

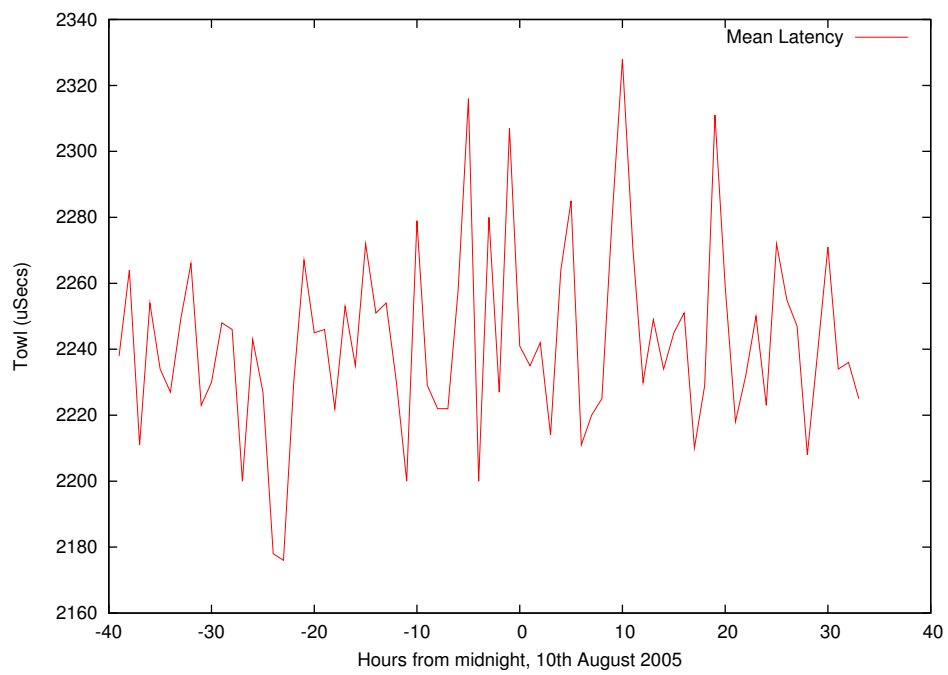


(a) Mean



(b) Minimum, maximum, and standard deviation

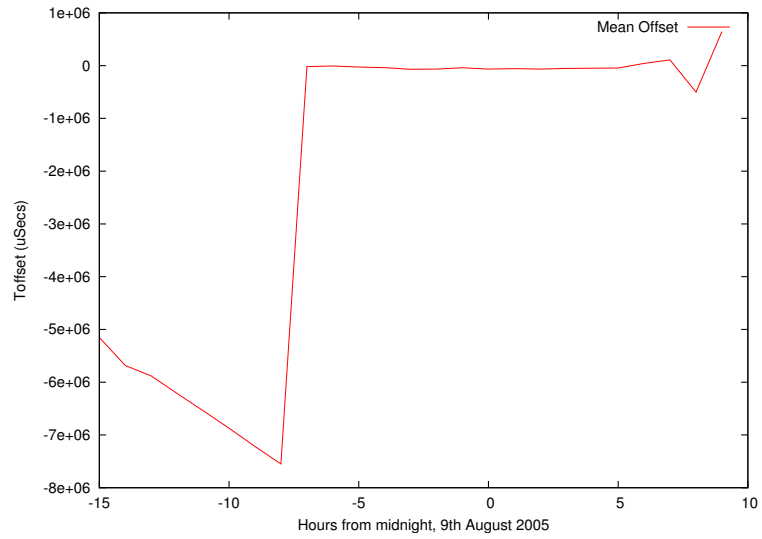
**Figure 9.15:**  $T_{owl}$  measurements, series 3



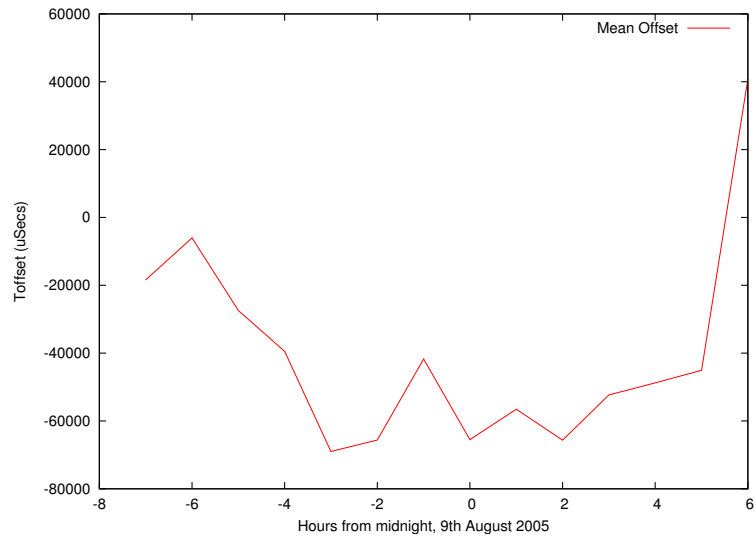
**Figure 9.16:** Mean  $T_{owl}$  values over 72 hour measurement period

Figure 9.17(a) shows the mean  $T_{offset}$  values of series 1. As can be seen the offset between the clocks of the hosts involved was initially quite high. It was discovered that the clocks were diverging over time during this period as NTP was not functioning correctly on the host located at TCD (the exact reason for this is unknown). Between 16:00 and 17:00 on August 10th the clock on the affected host was resynchronised with the NTP server, and the NTP daemon was restarted. This is shown in Figure 9.17(a) by the sudden change in the graph at -7 hours from midnight. Between approximately -7 and +5 hours from midnight the clocks remain synchronised, as can be seen in more detail in Figure 9.17(b). At the end of the first 24 hour period, however, the clocks began to diverge once again. This divergence can be seen to continue in the initial measurements of series 2, shown in Figure 9.18(a), until -10 hours from midnight (14:00), at which point the NTP daemon was once again restarted. From this hour until the end of series 2 the clocks remained reasonably synchronised as shown again in more detail in Figure 9.18(b). This continued for the start of series 3 until 15:00 (-9 hours) on the 11th of August, at which point the clock on the second host, located at QUB, lost synchronisation with the NTP server. The two clocks then continued to diverge for the rest of series 3, as seen in Figure 9.19. Figure 9.20 shows how the offset of the clocks varied over the full 72 hour period.

Clearly this bears investigation. Grid-Ireland are in the process of installing GPS-disciplined clocks at all their grid gateways, but they do need to establish why their NTP clients lose synchronisation.

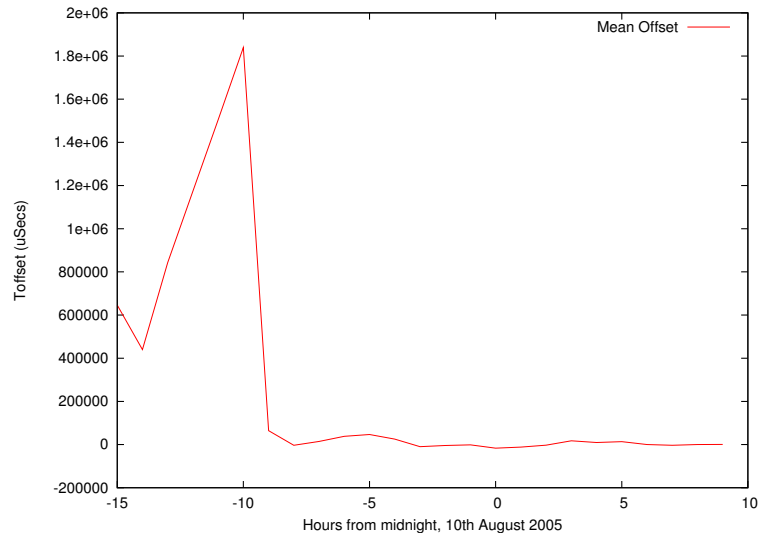


(a) Entire series

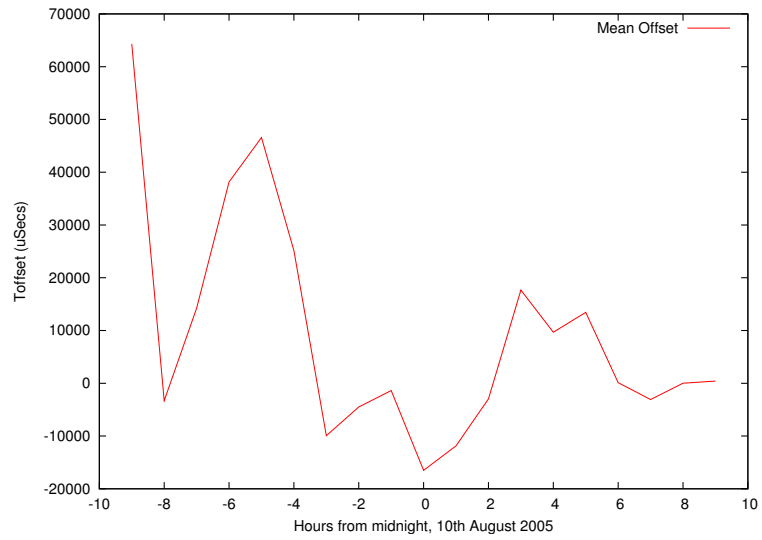


(b) After NTP restart

Figure 9.17:  $T_{offset}$  measurements, series 1

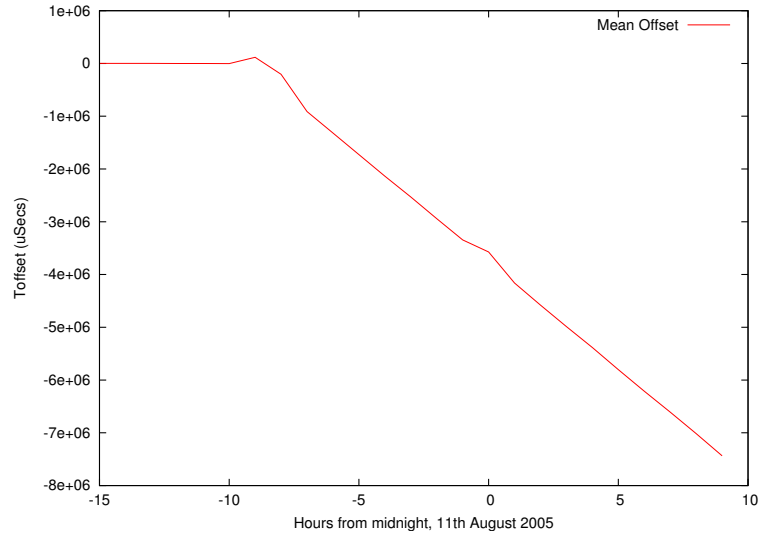


(a) Entire series

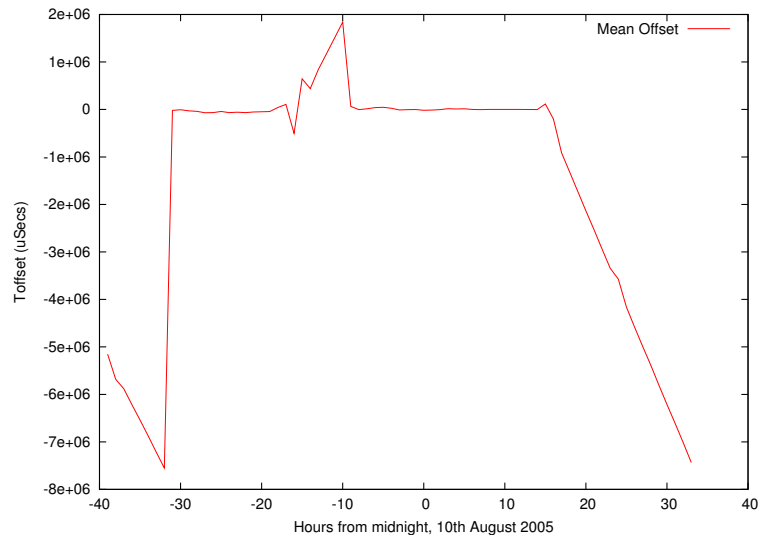


(b) After NTP restart

**Figure 9.18:**  $T_{offset}$  measurements, series 2



**Figure 9.19:** Mean  $T_{offset}$  values, series 3



**Figure 9.20:** Mean  $T_{offset}$  values over 72 hour measurement period

## 9.4 SCI TRACE ANALYSIS

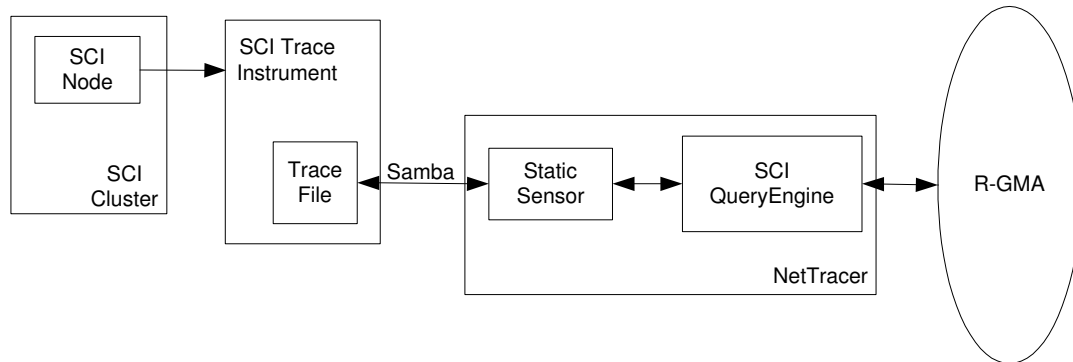
The aim of this experiment is to carry out statistical analysis on data obtained from non-invasive monitoring of the traffic generated by a SCI node in a cluster system. The traces to be analysed were taken from two SCI systems with different topologies, a 1-d two node ring and a 2-d torus with six SCI ringlets. The first trace was acquired using the SCI trace instrument developed in the Computer Architecture Group at Trinity College Dublin [24], whilst SCILAB's SCITRAC SCI trace instrument was used to obtain the second trace.

The purpose of the analysis is to generate statistics suitable for use with SCI simulation models. In [25] it was demonstrated how through the analysis of non-invasively obtained interconnect trace data, interconnect simulation models can be verified, and their parameterisation tuned, in order to ensure that they are an accurate representation of the real system. This was shown for a SCI node model developed by the University of Oslo [32]. By driving the node model with realistic load descriptions, derived through analysis of traces collected by the SCI trace instrument from real SCI systems, the model output could be compared with the output of the real system under study. Any mismatch observed would therefore allow for the fine tuning of the model parameters.

This experiment serves as an example of using the SANTA-G framework with a hardware instrument, and also demonstrates the SCI QueryEngine.

### 9.4.1 Configure the SANTA-G system

The deployment for the experiment is as shown in Figure 9.21. The SCI trace instrument will be used to collect the SCI traffic from one node in the SCI cluster into a raw trace file. Because the trace instrument generates a single trace file, rather than a dynamically generated set of files, the NetTracer *static* sensor can be used. Here the sensor's only purpose is to inform the QueryEngine of the file's existence, and to provide access to the file through its file server. The QueryEngine used is the SCI QueryEngine, as described in Chapter 7.



**Figure 9.21:** SCI trace analysis deployment

### 9.4.2 Write the Consumer code

The consumer is a modified version of a statistics tool (see Figure 9.22) developed by the author as part of [18]. The tool has been modified to use R-GMA Consumers to gather the trace data published to R-GMA by the SCI QueryEngine.

The consumer generates statistics for packet inter-arrival time, packet size, output throughput (bytes leaving a node’s output buffer), bypass throughput (bytes leaving a node’s bypass buffer), and node throughput (sum of output and bypass throughput). In [18] the inter-arrival time and packet size probability density functions (PDF) created by the tool were used to drive the University of Oslo’s SCI model. The throughput statistics obtained from the real trace were then compared to the simulated output in order to verify the model.

### 9.4.3 Run the experiment

To run the experiment first the trace file must be ‘published’ by starting an appropriately configured static sensor. The SCI traffic contained in the file can then be analysed by querying R-GMA using the modified statistics tool.

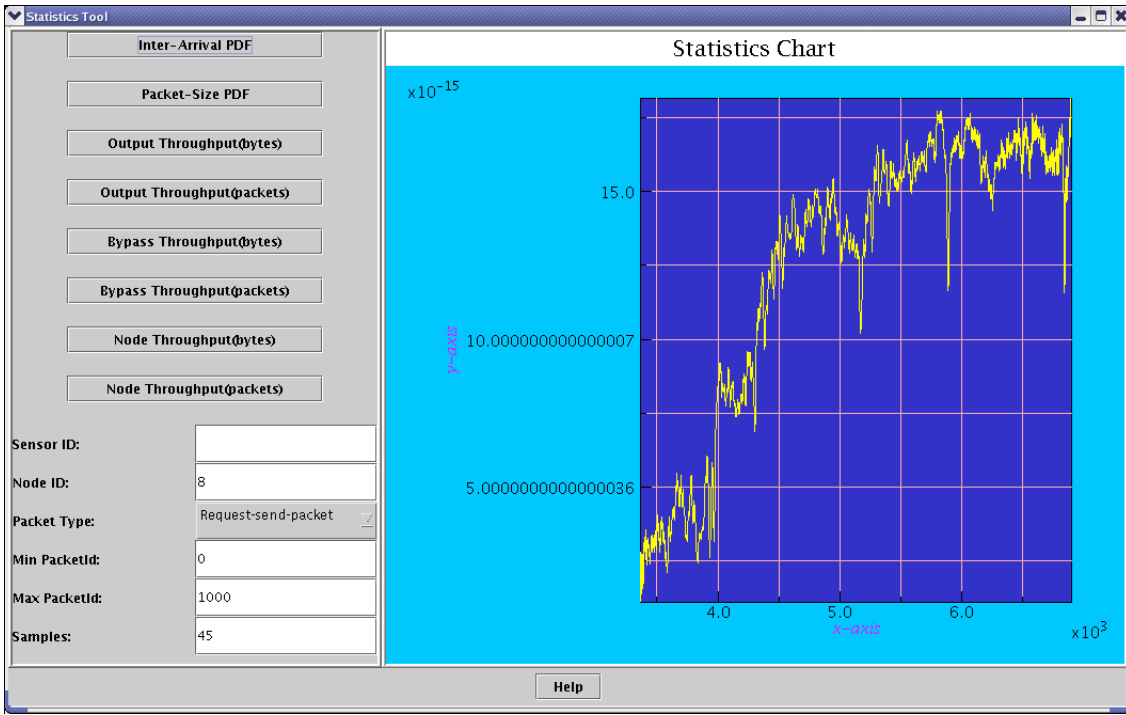
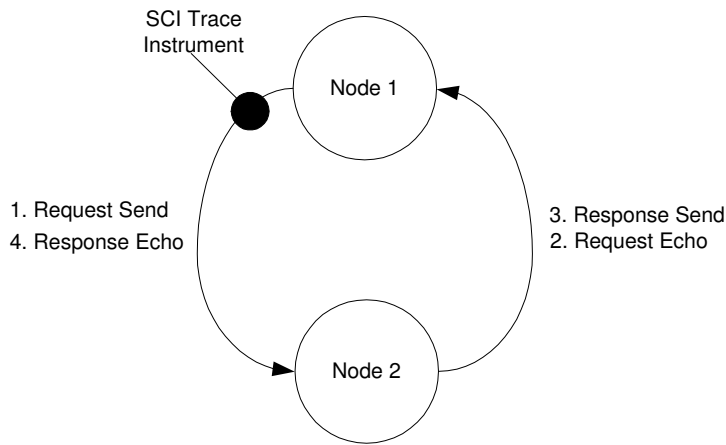


Figure 9.22: SCI trace analysis tool

### 1-d 2-node ringlet

For this experiment traffic was generated on the SCI links by *scibench*, a PCI-SCI performance tool. The SCI trace to be analysed covers a very short period of time, approximately 1.8ms. In this time 5,817 packets were collected and stored in the trace file by the trace instrument. The majority of the packets seen in the trace were either ‘request-send-packets with 64 bytes of data’ ( $\sim 52\%$ ) or ‘response-echo-packets’ ( $\sim 42\%$ ). The remainder were sync packets, used in the SCI protocol both during initialisation and normal operation to allow the receiver of the sync packet to check and adjust its circuit timing [1]. Because this trace was taken from the output link of an SCI node the request-echo and response packets generated by the second node are not seen, as these are absorbed by the target node (see Figure 9.23).

Figure 9.24 shows the probability density function calculated by the consumer for the



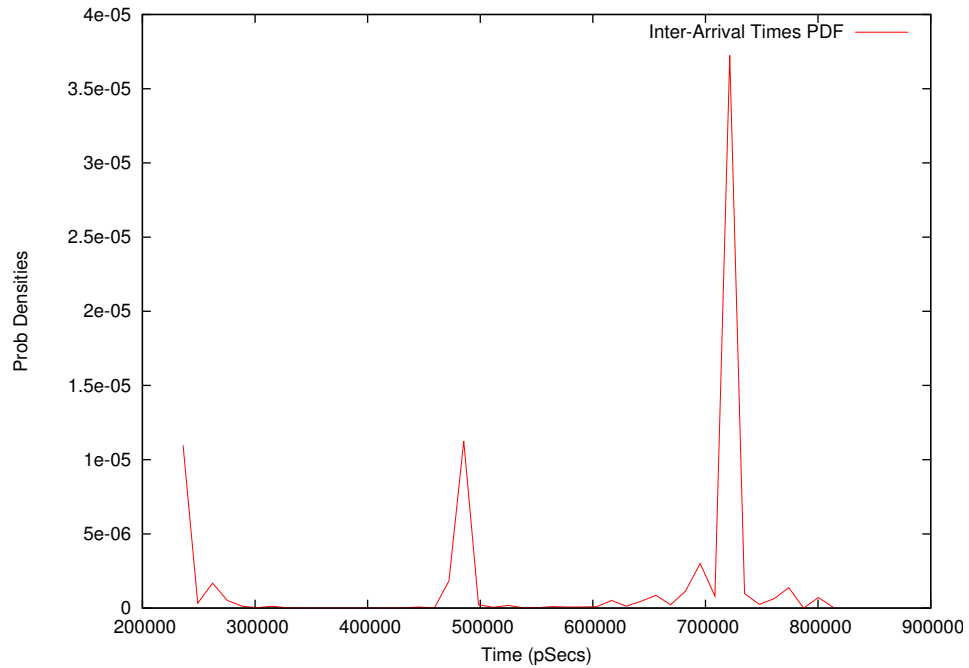
**Figure 9.23:** SCI transaction

inter-arrival time of packets generated by the target node. As can be seen the main peak in the graph is located from  $0.6\mu\text{s}$  to  $0.8\mu\text{s}$ . This means we can expect the majority of the inter-arrival times to be in this range, and this should be reflected in the model. A second peak can be seen from approximately  $0.45\mu\text{s}$  to  $0.5\mu\text{s}$ .

As stated the majority of the packets seen in the trace were request-send-packets with 64 bytes of data. These packets have a total packet size of 80 bytes, the SCI header (16 bytes) with 64 bytes of data. This is verified by the packet size PDF shown in Figure 9.25. The peak located at 6 bytes corresponds to the response-echo packets.

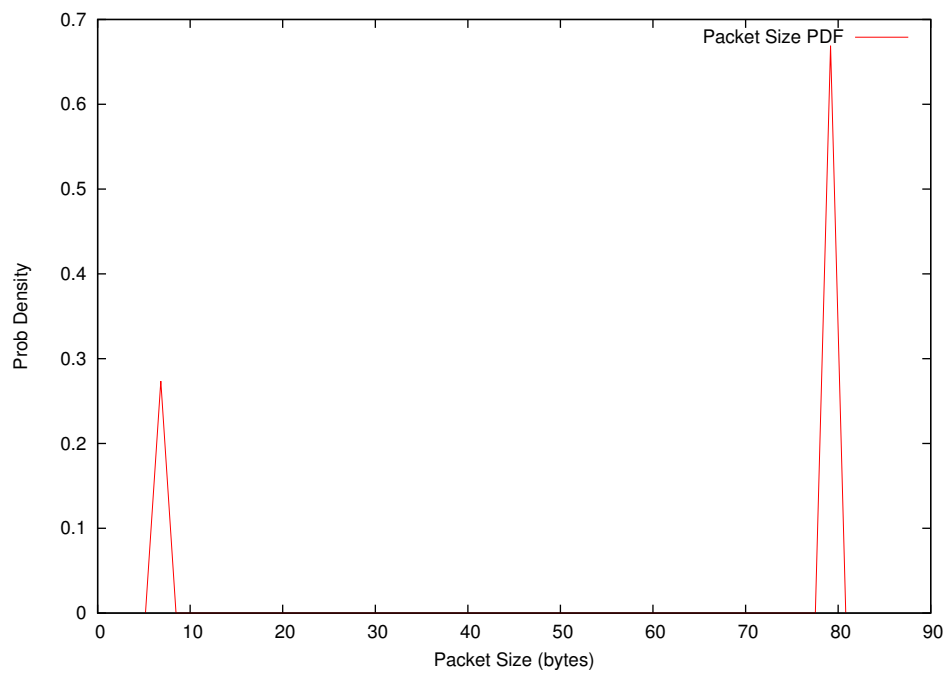
The graphs of output, bypass, and node throughput obtained during execution of the scibench benchmark are given in Figure 9.26. The mean output and bypass throughput over the full trace were 136 Mbytes/sec and 8 MBytes/sec respectively. As this is a two node ring the bypass throughput is calculated purely from the echo packets, as although these packets are generated by the target node they are placed in the bypass buffer for transmission. The mean node throughput, the total throughput of the target node, was approximately 142 MBytes/sec (the full available SCI bandwidth of 500 Mbytes/sec was therefore not used).

As can be seen in Figure 9.26 the throughput was not constant over the entire trace.

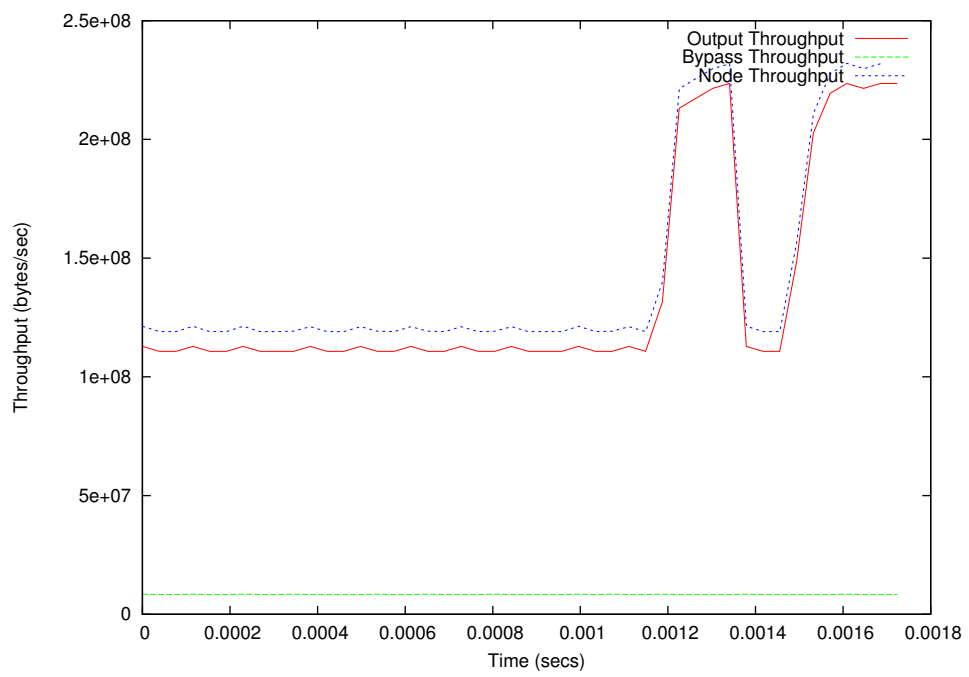


**Figure 9.24:** Packet inter-arrival time PDF

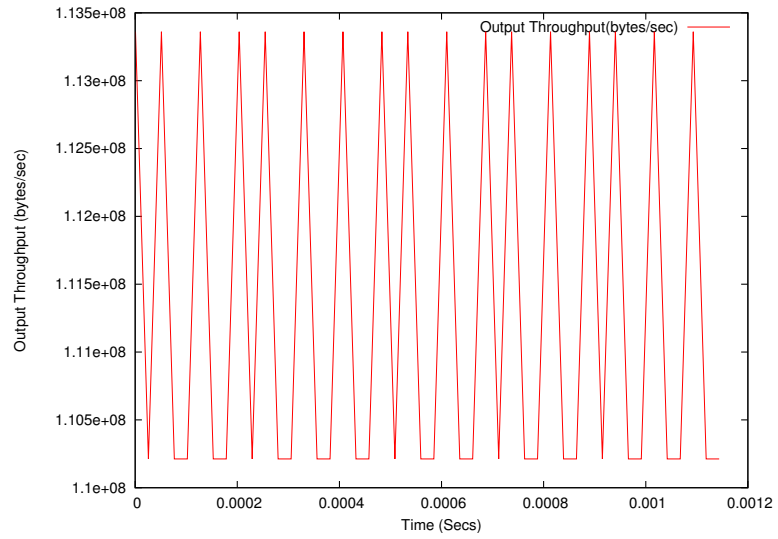
The mean throughput at the start of the trace was approximately 112 MBytes/sec (see Figure 9.27(a)). Towards the end of the trace, however, the throughput can be seen to increase to a maximum of 224 MBytes/sec (see Figure 9.27(b)), before falling briefly, and then increasing once again. Figure 9.28 plots the relative times for packets contained in the trace, i.e. the time between subsequent packets sent by the traced node. This graph clearly shows the two areas where the rate of packet generation increased (indicated by a decrease in packet inter-arrival times), thereby accounting for the two peaks seen in the node throughput.



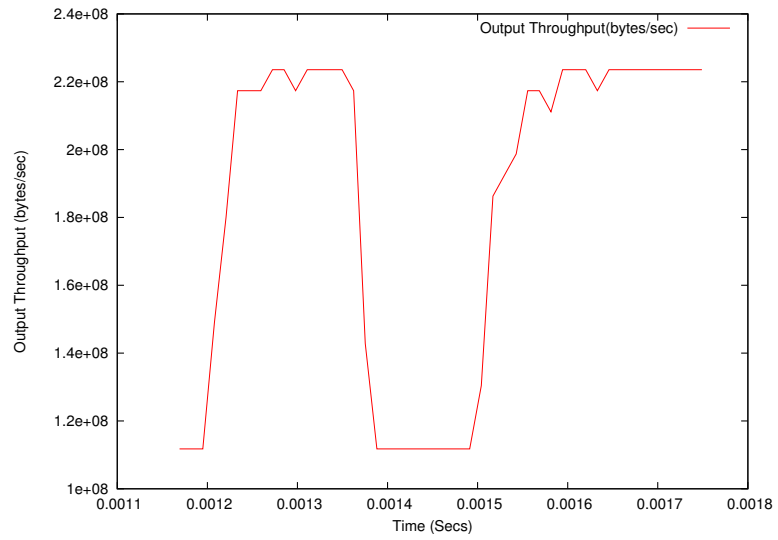
**Figure 9.25:** Packet size PDF



**Figure 9.26:** Throughput in bytes/sec

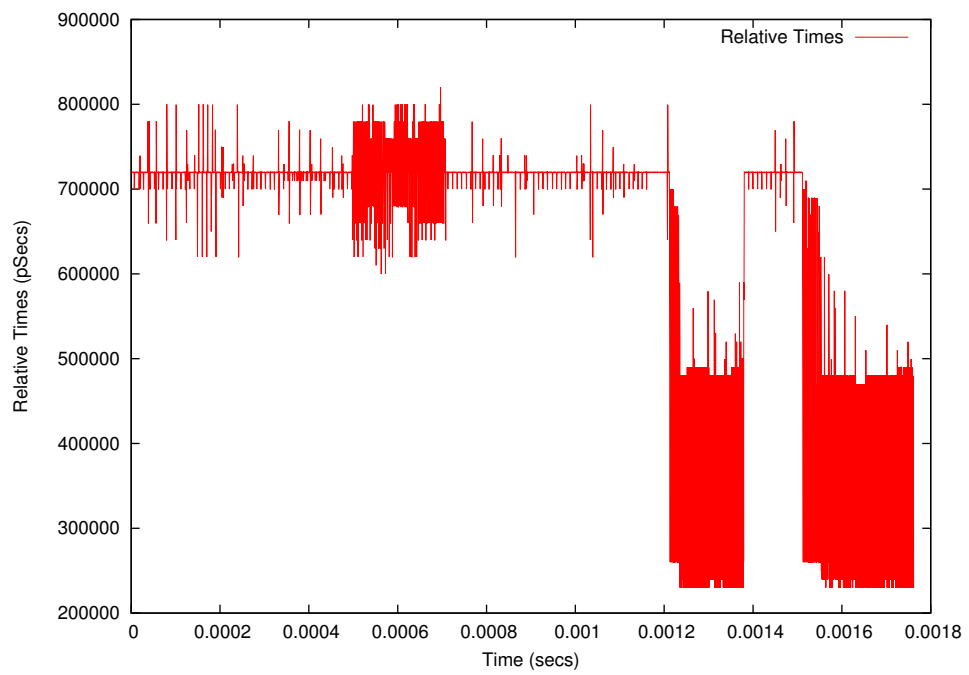


(a) Start of trace



(b) End of trace

**Figure 9.27:** Throughput in bytes/sec, start and end of trace

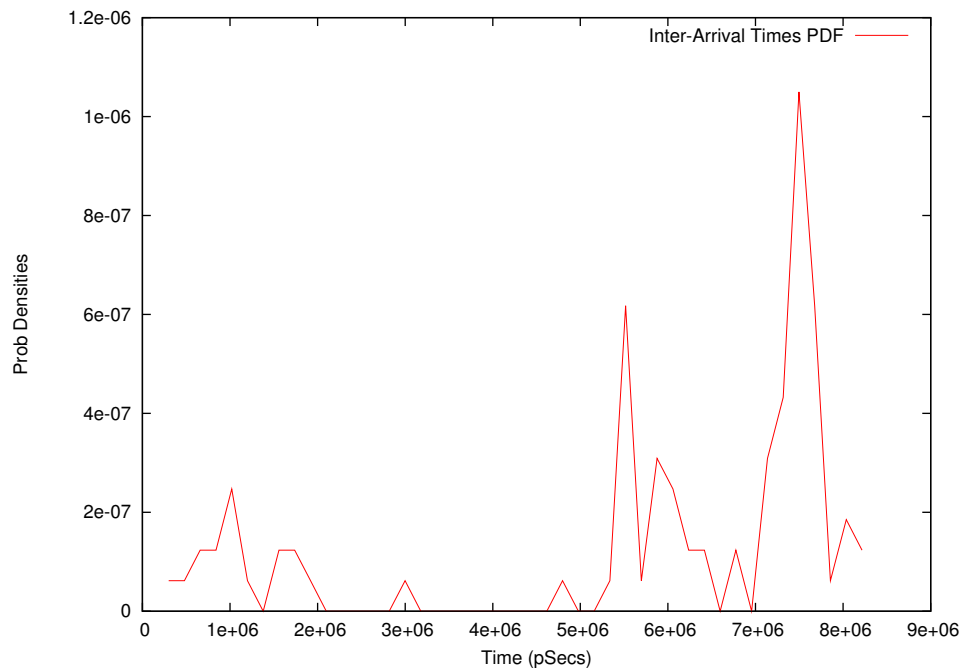


**Figure 9.28:** Packet inter-arrival time series

## 2-d 9-node torus

For this experiment the cluster system used was the Computer Architecture Group’s virtual reality engine (VREngine), an OpenGL engine that uses Chromium [38], a software system for interactive rendering on clusters of graphics workstations, to render 3-D graphical scenes generated by an application for display. The VREngine is composed of 9 nodes, connected using SCI interconnects, configured as a 2-dimensional torus with six SCI ringlets.

The SCI trace obtained covers a period of approximately 12 seconds, and contains 1,052 packets. Figure 9.29 shows the probability density function calculated by the consumer for the inter-arrival time of packets generated by the target node in this time.



**Figure 9.29:** Packet inter-arrival time PDF, 2-d torus

One of the most questionable (but almost universal) assumptions of the University of Oslo’s SCI model was that of a uniform ( $0^{th}$  order polynomial) distribution of addresses. Clearly this is untrue, as any reference to virtual memory working sets will confirm. The

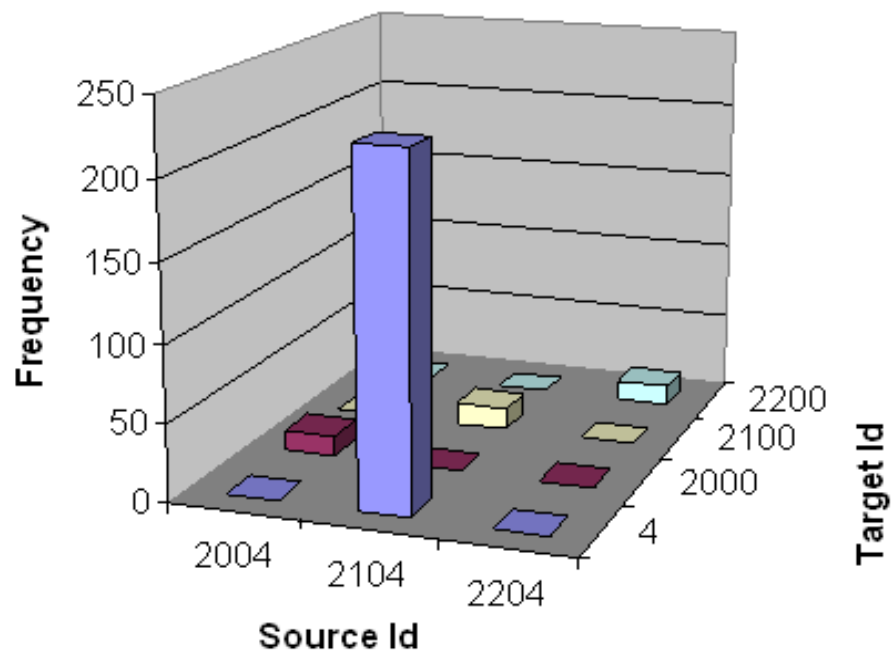
consumer was extended to generate statistics of the SCI address distribution. The distribution of target address accesses for the obtained trace is given in Table 9.9. Obviously each application will exhibit a different distribution.

Target Address	Frequency
0x00040000ff9a0140	179
0x0004ffffff000ff80	45
0x2000ffffff0000000	13
0x2100ffffff0000000	14
0x2200ffffff0000000	13

**Table 9.9:** Distribution of target address accesses

SCI uses 64 bit addressing, where the most significant 16 bits of the address specify the target node ID, and the remaining 48 bits give the offset of the memory address to access within the target node [1]. For the 2-d torus, table routing is used, where the upper half of the target ID is used to index the routing table. The 4 most significant bits select the routing table CSR register, and the remaining 4 bits select one of the CSR register’s 16 bits, the value of which determines whether the packet is to be switched or not [33]. Figure 9.30 shows the distribution of accesses versus source and target IDs as observed on the traced link. Only four non-zero values exist (representing five addresses) and while the reason is not known, one may speculate that this task farming application uses message passing between master 2104 and worker 0004 via a single message buffer at 0x0000FF9A0140, with some control exchanges at the other four addresses.

These are useful results for those directly concerned, and their usefulness is such that it is easy to lose sight of the fact that their acquisition is enabled by SANTA-G. It hardly needs to be said that these techniques are equally applicable to Myrinet and Infiniband interconnects, and their associated models, and that accurate modelling is extremely valuable for exploring large scale configurations for very large clusters that are too expensive to build and test.



**Figure 9.30:** Distribution of accesses versus source and target IDs

## Chapter 10

# GRID-WIDE INTRUSION DETECTION

A major application of the NetTracer system arose from within Grid-Ireland. Firstly there was a desire to instrument all sites to detect attempted security intrusions. Secondly it was felt necessary that all security alerts generated at sites within Grid-Ireland be visible at the Grid-Ireland Operations Center (OpsCentre). This was to provide the OpsCentre with an overall picture of the state of security of the entire Grid-Ireland infrastructure at any time, starting with intrusion detection. This model of grid-wide, non-invasive instrumenting and monitoring closely matches that of the SANTA-G framework. Consequently the NetTracer was extended for the Snort network intrusion detection system. This work led to the design of the grid-wide intrusion detection system (GIDS) [19][20] that is described in the following chapter.

## 10.1 INTRUSION DETECTION

Detecting attacks on computers or networks, and computer misuse from either inside or outside a network, is known as intrusion detection. Intrusion detection systems (IDS) provide three main functions: monitor, detect, and respond [15]. Policies are defined by the IDS administrator in the negative, i.e. those events that should NOT be seen on the network. The IDS monitors the network, or individual computers, and if one of the defined events is detected then the system responds by issuing an alert to the responsible higher-level system or person.

There are two main types of IDS, host-based, and network-based.

### Host-Based IDS

Host-based IDS (HIDS) will only protect the host system. This is done by either monitoring the network traffic received/sent by the host, or by monitoring the services and system files. By monitoring system files it is possible for host-based systems to detect unauthorised file modification (e.g. using Tripwire) that could indicate a compromised system, or an attempted unauthorised activity from a user. Host-based systems can also analyse the network traffic for the host by collecting only packets addressed to the host, or sent by the host. This allows for the IDS to be configured to look for traffic specific to the services running on the host.

### Network-Based IDS

Network-based IDS (NIDS) will monitor the network traffic seen on the entire network segment they are connected to, rather than only the traffic addressed to a specific host. Packet-sniffing is used to collect all the network packets seen. These packets are then checked against the IDS policies for the network to determine whether they are acceptable. Alerts are generated for any packets that match the signature of possible malicious activity. Network-based systems can be used to detect attacks such as denial of service originating from outside the

network. Whereas a host-based system will be effective at detecting a successful compromise of a system, a network-based system will detect the initial access attempt.

A third type of IDS is a Distributed Intrusion Detection Systems (DIDS) [10]. In these systems multiple IDS distributed across different networks will co-operate to report detected alerts to a centralised location.

## 10.2 EXISTING APPROACHES

A drawback with NIDS is that they can only see traffic on the network segment that they are connected to. This can make it difficult, if not impossible, to detect large scale attacks on an infrastructure that spans multiple networks. Having separate IDS installed at each site also implies the need for a separate analysis team at each site, increasing cost and complexity. It can also slow down the detection and response to an attack. The collection of dispersed information is vital in intrusion detection. In order to detect a new attack or exploit effectively the analysts must have access to the data as quickly as possible. DIDS has been developed to solve these issues.

There are currently several systems deployed that implement a DIDS. All these systems follow a similar architecture of multiple distributed IDS reporting detected alerts to a central server. The IDS deployed can be either a NIDS or HIDS, or more commonly both. The alerts can be sent to the central server using either an automated system or manually uploaded by the site administrator. The alerts can be sent in ‘real time’ as they are detected, or possibly collected and uploaded as a batch job at regular intervals. At the central server usually some aggregation of the alerts and analysis is then done in order to generate some statistics on the types of alerts being detected and to identify any threats. This information is then made available to the sites participating in the system in the form of either further alerts or often through a web interface.

As stated there are currently several examples of this approach. DShield is an internet

based system that collects logs from firewalls in order to try and detect ‘trends in activity and to develop better firewall rules’ [39]. Logs can be submitted by any registered user, using either client software to automate the process, or manually via a web form. Reports of attacks are sent to the ISP from which the attack originated. A recent system (at time of writing) is the Fingerprint Sharing Alliance (FSA) [2]. This ‘is a coalition of telecommunications companies around the globe that are stamping out cyber attacks that cross company boundaries, continents and oceans’ [2]. The FSA uses proprietary software to collect traffic on a service provider’s network. Whenever activity is detected that is abnormal (this is determined by comparing the activity to previous ‘normal’ traffic patterns) the service provider is alerted. The provider can then decide whether the activity is malicious. If so then a ‘fingerprint’ of the activity is generated that is then shared in order that the other members of the alliance can detect any further occurrences of that activity.

In [22] a system is proposed that utilises a ‘grid’ model for performing intrusion detection. The main goal of the system is to detect denial of service (DoS) and distributed denial of service attacks (DDoS). They argue that a single NIDS could lose its detection capabilities during attacks such as these due to the high volumes of network traffic experienced, since a NIDS could begin dropping packets if the load became such that it could not process the traffic quickly enough. Also the load on the host, in terms of CPU usage, could become very high and ultimately the machine could be rendered unresponsive. The solution they describe is to use a group of other nodes to perform the intrusion detection analysis. The role of a NIDS is replaced with that of a *dispatcher*. One or more dispatchers use Tcpcmdump to continuously collect short periods (2 seconds) of network traffic into log files. Once a file is created the dispatcher contacts a *scheduler* in order to be assigned a *detection node*, i.e. a node that will do the analysis. The scheduler chooses the ‘best’ detection node to perform the analysis and also ensures the load is distributed evenly across the group of detection nodes. The dispatcher then transfers the file to the selected node using Globus GridFTP and starts the detection process by using ‘globus-job-run’, hence the ‘grid’ orientation. Upon receiving

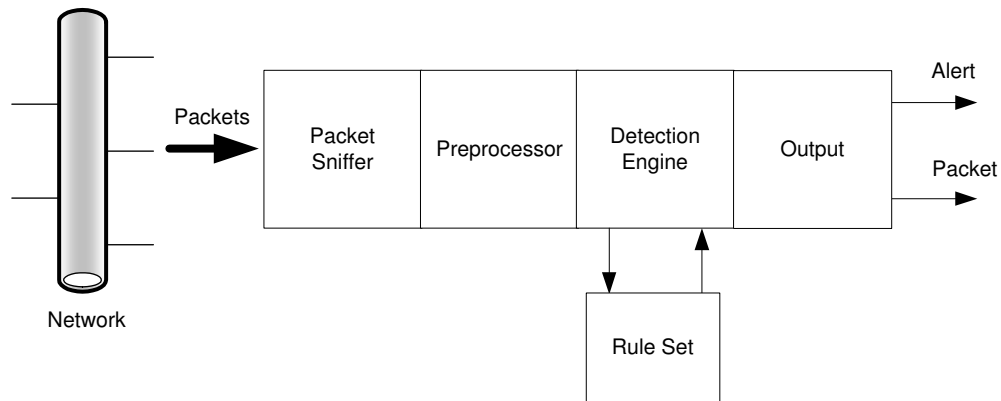
a trace file the detection node analyses the packets contained in the file using custom attack detection algorithms to determine if an attack is taking place. If an attack is detected an entry is made in a ‘Block List Table’ that is stored in a central database. A record for any suspect packets that *may* indicate an attack are stored in a ‘Suspect Table List’. A second phase of detection is performed on these tables by a ‘*chronic detector*’. The chronic detector uses the same detection algorithms as the first phase of detection but looks for attacks over a longer duration, 10 to 20 seconds, as opposed to the initial 2 seconds analysed in the first phase, i.e. it looks for chronic attacks. Although this system is aimed at using a grid model for intrusion detection, rather than providing an intrusion detection system for the grid, it could be adapted for the latter purpose.

Intrusion detection is a vital task for those administering the grid. A grid is a large scale distributed system with sites in many organisations and geographic regions. Each site will have its own security policies and procedures. The grid will only be as secure as its weakest link. To secure the grid it is important that security information be available to site administrators in a timely and efficient way. Relying on individual site administrators to share information would introduce critical delays in the process and hence a coordinated approach is preferable. The following section describes the design of a Grid-wide Intrusion Detection system (GIDS) using the SANTA-G NetTracer and Snort.

## 10.3 GRID-WIDE INTRUSION DETECTION

### 10.3.1 Snort

Snort is an open source network intrusion detection system capable of performing real-time traffic analysis and packet logging on IP networks [5]. Snort has become the de-facto standard for intrusion detection and prevention [42]. It was designed to be lightweight and compatible with multiple operating systems, with 4 main components (see Figure 10.1):



**Figure 10.1:** Snort architecture

**A packet sniffer** to trace packets from the network.

**A preprocessor** to determine if a packet requires further analysis and to format it for the detection engine, e.g. to normalise a HTTP request string in a packets data payload.

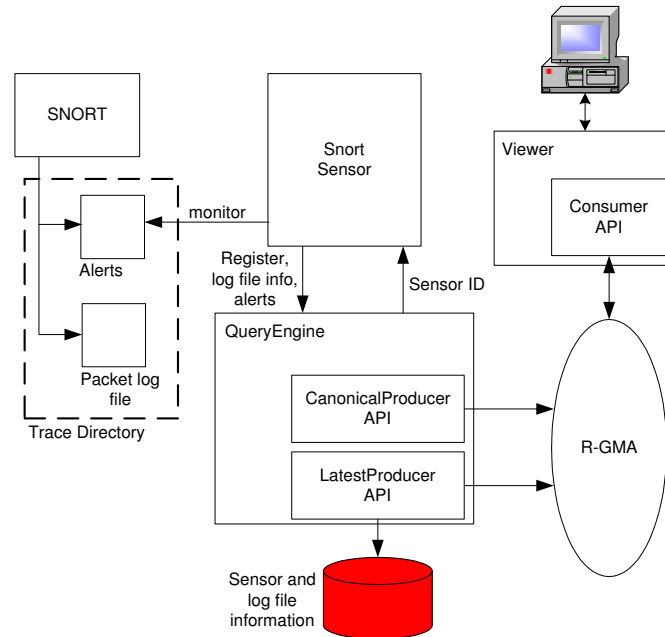
**A detection engine** that compares each packet passed to it from the preprocessor against a set of defined security rules.

**An output module** that generates an alert if a match is found by the detection engine (i.e. a suspect packet has been found). The output module can be used to output the alert in several different ways, e.g. to a log file, a SQL database, a Windows Popup, a UNIX socket. The full packet that triggered the alert can also be logged, again in a number of formats, including a Tcpdump compatible binary log file.

### 10.3.2 NetTracer Snort sensor

In the GIDS, Snort is to be used to monitor nodes in the system. Snort will be configured to log alerts to a log file as they are detected. The packet that triggered the alert will also be logged in a separate log file. This packet log file follows the same format as Tcpdump, so the log files created by Snort are compatible with the NetTracer. In order to extend

the NetTracer for Snort an additional sensor type had to be added, the *Snort sensor* (see Figure 10.2).



**Figure 10.2:** NetTracer Snort monitoring

The Snort sensor monitors the alerts file, and when a new alert is detected its details are sent to the QueryEngine, which records the alert, and then publishes it to the R-GMA using a *StreamProducer*. Users can view these alerts by using the NetTracer Viewer GUI, or by querying the R-GMA directly. The schema for the Snort alerts table is shown in Table 10.1 (where PRI denotes a primary key).

The full packet data of the packet that triggered the alert can then be viewed by querying the packet log file generated by Snort. This is accomplished using the same mechanisms as for querying the log files generated by Tcpcdump (described in Chapter 7).

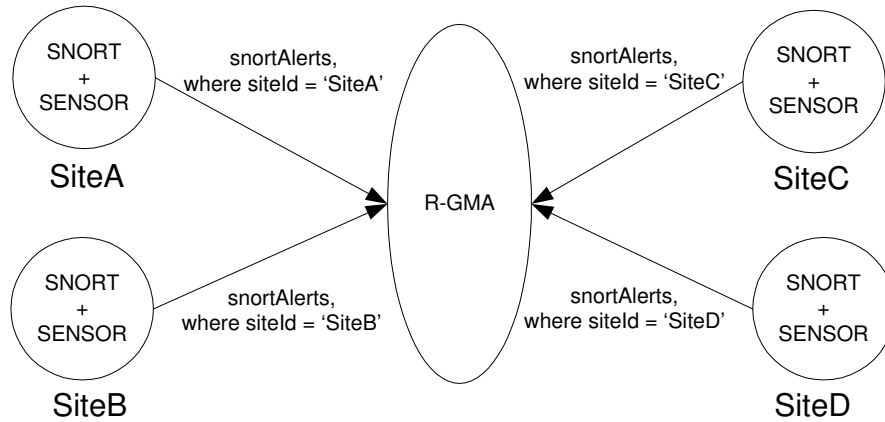
Field	Key	Description
siteId	PRI	Site ID
sensorId	PRI	Sensor ID
fileId	PRI	Log file ID
alert_timestamp	PRI	Timestamp of when the event was logged
alert_type		Type of event
generator_id		Specifies the generator that detected the alert
signature_id		Identifies the rule that was used by the generator
revision		Specifies the revision of the rule used
message		Alert message
classification		Alert classification name
priority		Alert priority
source_ip		Source IP address
destination_ip		Destination IP address
source_port		Source port
destination_port		Destination port
protocol		Packet protocol
data		Packet header data
info		Any additional information included in the alert

**Table 10.1:** Snort alerts table schema

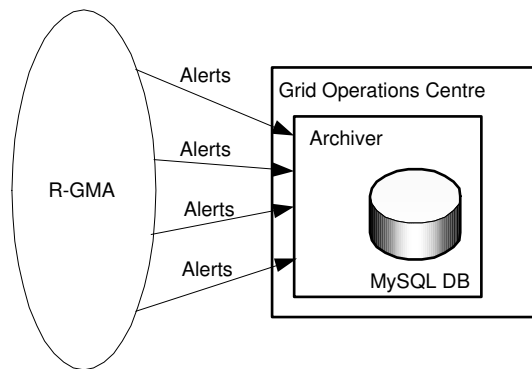
### 10.3.3 GIDS Design

It is intended in the design of the grid-wide intrusion detection system that each site within the grid will run a set of Snort sensors that will publish any logged alerts to the R-GMA as described above. This will result in each site streaming alerts into the information system (see Figure 10.3).

In order for these alerts to be collected at the OpsCentre a second R-GMA component, an Archiver, will be used. An Archiver aggregates streams from multiple Consumers and stores the information into a MySQL database. It then republishes this information using its own Producer. By running an Archiver at the OpsCentre that will query for the alerts being published by the Snort sensors, all the alerts can be aggregated to form a ‘grid-wide intrusion log’ (Figure 10.4).



**Figure 10.3:** Multiple sites stream alerts to the R-GMA



**Figure 10.4:** Archiver collects alerts to grid-wide intrusion log

To create the Archiver for the intrusion log, first a producer must be instantiated that can republish the aggregated alerts. Because it is required that all the alerts be stored persistently, a `DataBaseProducer` is used:

```
DataBaseProducer alertsDB = new DataBaseProducer(
    "jdbc:mysql://localhost/intrusionLog",
    "user", "passwd");
```

This producer is then passed to an Archiver:

```
Archiver intrusionLog = new Archiver(alertsDB);
```

The table to be archived is specified by calling the Archiver's `declareTable` method:

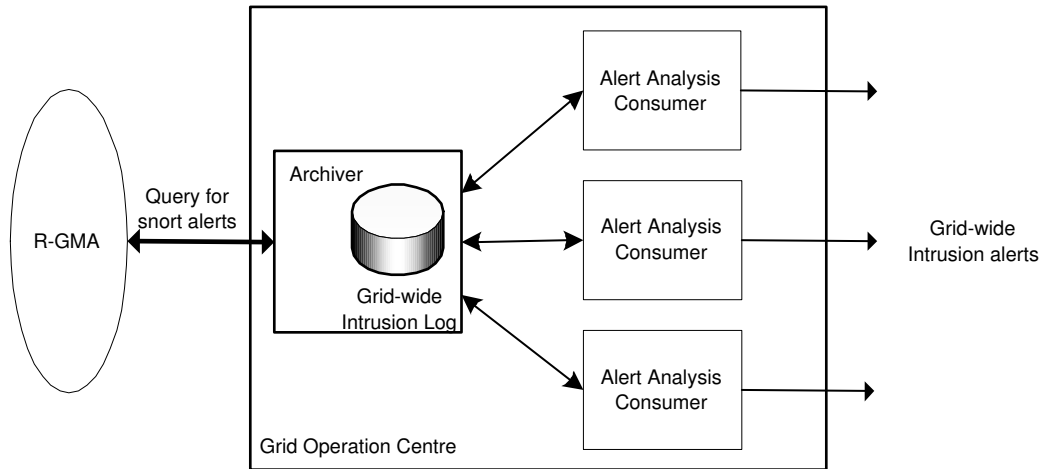
```
intrusionLog.declareTable("snortAlerts", "",  
                           "CREATE TABLE snortAlerts (" +  
                           "siteId VARCHAR(100) NOT NULL, "  
                           ...
```

Once this is done the Archiver will create a Consumer that will continuously query all registered producers of the `snortAlerts` table. The Consumer will also be informed if new producers of the table, i.e. new Snort sensors, become available. Alerts received by the Consumer are passed by the Archiver to its `DataBaseProducer`, which will both store and republish them.

The intrusion log in itself forms a valuable source of information to the analysts at the OpsCentre. As well as allowing for the tracing of attacks across the entire grid infrastructure, it also allows for common analysis tasks, such as aggregation, to be carried out simply by submitting SQL queries. Aggregation is the task of grouping alerts together using various parameters such as source IP or destination port.

The intrusion log represents the first level of the GIDS, the raw data collected from the sites that make up the grid infrastructure. The next level will filter and analyse the alerts published in order to detect patterns that would signify an attempted distributed attack on this infrastructure. Because it is expected that a significant number of alerts will be generated and logged it is intended to automate this analysis as much as possible. To do this a high-level incident detection, tracking and response platform will be created by using custom coded Consumers (as shown in Figure 10.5) and Archivers. These Consumers and Archivers will query the log for specific predetermined alert patterns. This can be done using a *Continuous* query. This means that after submitting the query the Consumer or Archiver will continue to receive the results as they are published by the grid-wide intrusion log's Producer. They

will then analyse the alerts, possibly by using AI techniques such as adaptive neural nets and category theory, to determine if they represent a possible malicious attack. If so they will then generate their own alerts that will be issued to the responsible person or higher-level system.



**Figure 10.5:** Intrusion log analysis by alert Consumers

The work on the alert Consumers is in its preliminary stages and is expected to represent the majority of the future effort on the system. An initial, very basic, example implementation is discussed in the following chapter in order to illustrate the structure of the analysis process.

It is intended to complement the Snort data with information from other security components. Work is currently underway creating new sensor types and query engines for use with such tools as Tripwire and AIDE (Advanced Intrusion Detection Environment) and Nagios. It is also proposed to investigate more active security measures, i.e. measures that track grid interactions and respond to unwanted security events in an adaptive way. This will require both extended security monitoring and tracking, and the automatic coordination of this across a number of security tools and all sites involved in the events. A minimum set of recognised events can be derived from the proactive filtering of log files resulting from normal grid activity, the regular probing of the grid infrastructure by tools such as the HostCheck

test suite developed by the CrossGrid project, the Site Functional Tests (SFT) developed by the LCG/EGEE projects, and the standard and custom Grid-Ireland checks done for Nagios, and also from node-specific security measures such as Tripwire and AIDE. Auditing of relevant actions will be integral. The intent is to track and resolve security issues as they occur. While resolutions can be added in an ad-hoc manner, it is better that they be based on a unified management model. The management model can, for example, be based on the concepts of event, condition, action, where actions are conditioned by policies. Particular attention will need to be paid to formally defining this model to ensure that consistent local and global policies are easily attained.

## 10.4 DISCUSSION

The system described above has a number of benefits, many of which are derived from the use of a grid information system as the underlying transport mechanism. By using R-GMA there is no need to construct communication channels between the remote sensors and the central management server at the OpsCentre, as would be necessary with other approaches, and furthermore the characteristics of the streaming mechanisms of R-GMA closely match the desired alert channel properties. Once the alerts are published by the QueryEngines to the R-GMA they are automatically available to the grid. Secure communications between the sensors and the OpsCentre can be accomplished by configuring the R-GMA to use HTTPS. Collecting alerts at the OpsCentre from the distributed sensors is also a simple task thanks to the use of the Archiver component. This has the added benefit of being backed by a RDBMS which allows for data analysis and persistent storage. Another issue in existing systems is that the central server presents a single failure point, and also a possible target for attack. It is possible with GIDS to provide redundancy and replication of the intrusion log by running an Archiver at one or more other locations. Once started these Archivers will automatically begin aggregating published alerts. The addition of new sensors, or new sites to the system

is also handled by the underlying R-GMA mechanisms. New producers of alerts will register with the R-GMA at start-up and begin publishing alerts. The Archiver gathering the alerts will be notified of the new producers and will immediately begin collecting alerts from them, with no change to the Archiver code.

The system also provides near to real-time logging of alerts to the OpsCentre. There will be only a minimal delay between the detection of the alert at a site and the appearance of the alert in the intrusion log at the OpsCentre. This allows for the immediate sharing of data and collaborative analysis. This will significantly reduce the time from alert logging to attack detection.

How does GIDS compare with the existing state of the art, e.g. the system described in [22] by Fang-Yie Leu et al. Firstly, GIDS is explicitly designed for grid-wide operation whereas the latter is intended for a single ‘network management unit (NMU)’, e.g. an enterprise’s intranet or college campus. The GIDS Snort sensors are analogous to the combination of *dispatchers* to collect traffic, *detection nodes* to perform the intrusion detection analysis, and a *scheduler* to assign detection nodes and to balance the load across them. The system described by Fang-Yie Leu et al does not include a centralised database and as such there is no direct comparison with the grid-wide intrusion log of GIDS. As the result of the analysis for each NMU is stored in a separate database local to the NMU, one could replace each database with a single centralised database, but this would result in the system suffering from the same drawbacks as those for other DIDS. Communication channels would have to be created between the detection nodes and the central database server, which in turn again becomes a single point of failure (whereas the GIDS grid-wide log can easily be replicated using the R-GMA). It is also unlikely to be economically acceptable to dedicate groups of nodes at each site purely for the purposes of intrusion detection. Another major factor that limits its effectiveness is the use of custom analysis code for the detection of attacks. Whereas GIDS uses Snort, a well-supported and proven public domain IDS with an active community constantly developing new rules to detect new threats, to perform the detection analysis,

Fang-Yie Leu et al have developed custom filters and algorithms for detecting attacks. The system is also heavily biased towards the detection of denial of service type attacks. In reality there are a vast number of other attacks that an IDS needs to be capable of detecting in order to be effective.

## Chapter 11

# INTRUSION DETECTION EXAMPLE

An attack often has two phases, a reconnaissance phase, followed by the actual exploit. The reconnaissance phase can itself have two parts, network mapping and host mapping [30][29]. Network mapping is an attempt by the attacker to determine the hosts that are available within a network. By doing this they can exclude the IP addresses of hosts that do not exist from the network address range they are targeting. Once the host addresses are known the next stage is to obtain information about the hosts, such as the OS type and the services the host is running. This allows the attacker to target the host with an exploit known to work against a particular OS or service. Because the time between the reconnaissance phase and the launching of an exploit against a vulnerable service can be short, it is important to monitor and detect these reconnaissance attempts.

As described in the previous chapter the GIDS has two levels. The first level gathers the alerts from the sites in the grid to form the grid-wide intrusion log. The second level analyses the intrusion log to detect attacks on the grid infrastructure and generates alerts when attacks are detected. The following describes three example intrusion log analysers

that each use a different method to detect one form of attempted reconnaissance of the grid infrastructure, systematic multi-site port scanning.

## 11.1 TEST DEPLOYMENT

In order to test the GIDS in a grid environment TCD's TestGrid infrastructure was used. TestGrid makes use of virtual machine and networking technology to provide a complete and faithful replica of the Grid-Ireland national infrastructure. Replicas of three Grid-Ireland sites were used for the test: csTCDie, giDITie, and giDCUie. A replica of the central Grid-Ireland R-GMA registry was used by the QueryEngine deployed on each replica site, as well as by the Archiver used to collect the alerts published by each site. The R-GMA server also hosted the intrusion log analysers. Figure 11.1 summarises the test deployment.

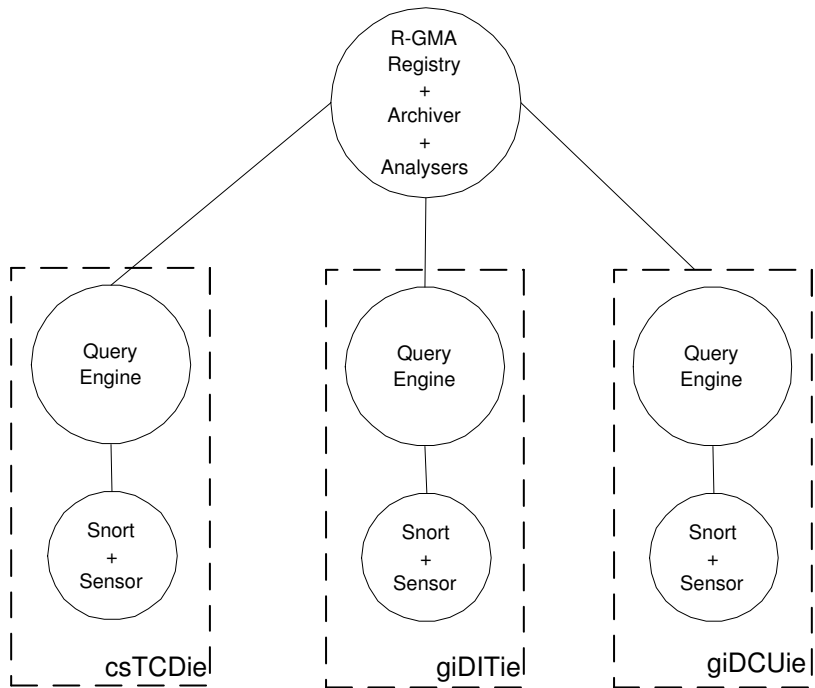


Figure 11.1: GIDS test deployment

## 11.2 EXAMPLE ANALYSERS

Port scanning is an attempt to determine the services that are active on a host. To perform a port scan an attacker sends packets to the ports of interest on a host and collects the responses. From the responses received the accessible services on the host can be identified. A set of rules along with a preprocessor module for detecting port scans have been defined for Snort by members of its user community. The majority of attempted port accesses in a port scan will result in a negative response from the targeted host, i.e. no service is available on that port. A large amount of negative responses in a short period of time is therefore used by the sfPortscan preprocessor to detect a port scan. When a portscan is detected the preprocessor generates an alert and logs a *pseudo* packet to the packet log file. The payload of the pseudo packet is used to store information about the detected scan, such as connection count, port count etc. The example intrusion log analysers described below each use a different analysis method to detect an attempt to map the grid infrastructure.

Any intrusion log analyser needs to perform the following basic operations:

1. Log filtering
2. Pattern Matching
3. Alerting

First the log is filtered for the alerts of interest. This can be done using continuous consumers as described in the previous chapter. With a continuous consumer the alerts that satisfy the query are pushed to the consumer as soon as they are published. Some form of pattern matching must take place in order to determine if the alert stream indicates a possible attack. If so then an alert will be generated.

### 11.2.1 Simple Patten Matching Analyser

The first analyser is an example of very simple pattern matching. This method is comparable to the most basic approach used by Snort, where a 'signature' that identifies an attack is defined, packets are examined to determine if they contain the signature, and if so an alert is logged. The benefits of this approach is that it is relatively simple to implement and also reliable, in that it will always alert if a defined signature is detected. The difficulty is that only attacks for which a signature has been defined can be detected. Also it may be possible to evade detection by modifying the attack so that it no longer matches the signature.

The analyser uses this approach to detect a portscan across multiple hosts and/or sites in the grid. To do this first the grid-wide intrusion log must be filtered for portscan alerts. All alerts logged by Snort contain signature information. This information describes the generator that detected the alert, the specific rule that was used, and the revision of the rule used. This information is published by the QueryEngine in the `snortAlerts` table as three separate columns, `generator_id`, `signature_id`, and `revision`. This allows for the searching of alerts based on the preprocessor that generated them. In the case of the `sfPortscan` preprocessor the generator ID used is 122. This can be used to filter the logged alerts. The filtering of the log in this case is very simple. A continuous Consumer is used to query the log for any published port scan alerts.

```
Consumer portScanConsumer = new Consumer("SELECT *
FROM snortAlerts
WHERE generator_id = 122",
Consumer.CONTINUOUS");
```

The above statement creates a Consumer that will return any alert generated by the `sfPortscan` preprocessor published to the intrusion log.

When an alert is received its details are extracted from the result set and sent to a pattern matcher to determine if an alert should be generated. Again the pattern is very simple. If we

see a portscan alert at more than 1 site from the same source IP then a ‘grid infrastructure portscan alert’ should be generated. Once the alert is triggered a ‘grid alert’ email is created and sent.

Obviously a portscan alert at a single site would warrant further investigation in itself. Single site alerts can, however, be seen by directly querying the intrusion log. The purpose here is to illustrate how patterns of alerts across sites can be detected.

### **11.2.2 Heuristic-based Analyser**

In this example, rather than looking for a specific pattern of alerts, an algorithmic logic [34] is applied to determine whether an alert should be triggered. The Snort portscan preprocessor, described above, is an example of this approach. If a host is seen to generate RST packets above a certain threshold, within a fixed period of time, then the preprocessor decides that a portscan is occurring. This type of approach allows for the detection of activity that does not have an identifiable signature. It may also detect new types of attacks for which a signature has not yet been defined. The difficulty with this approach is in setting the threshold at the correct level, such that false positives are minimised, whilst still retaining the sensitivity required to detect possible attacks.

The analyser being considered here bases the decision to generate an alert on a specific characteristic of the intrusion log, alert inter-arrival time, i.e. the time between alerts being published to the log. An attacker may attempt to mask their activity by flooding a network with packets known to trigger Snort alerts. Tools, such as Snot and Stick, can be used to generate such packets from a Snort ruleset. The large amount of false positives generated make it difficult to detect or trace actual attacks. The assumption with this analyser is that a sudden increase in the rate of alerts published to the log within a certain period could indicate that a hidden attack is taking place. Again a continuous consumer is used to collect alerts as they are published. Here, because a specific pattern is not being searched for, no

filtering is applied. The consumer stores the time that the alert was received as the ‘detect time’ of the alert. At regular intervals the consumer then calculates the average inter-arrival time of alerts since the last sample. If the time calculated is below a specified threshold then an alert should be generated. The intention is to tune the threshold value and sample interval based on observation of the intrusion log over time. Possibly an adaptive constant false alarm rate (CFAR) loop might be used to automate the tuning.

### 11.2.3 Anomaly-based Analyser

The final example uses an ‘anomaly-based’ approach. With this method the goal is to generate an alert whenever activity is detected that deviates from the normal. The heuristic-based approach above could also be described as anomaly-based, where normal activity is described by the discrete threshold and sample period. Here though, the idea is to determine anomalous behaviour through comparison with previously observed activity. Again, as with the heuristic approach, the benefit is that this method will be capable of detecting attacks for which signatures do not exist. Alerts when generated, however, will only indicate that some unusual activity has occurred, not that a specific attack has taken place. What the activity represents would not be immediately known, and would require further investigation. It can be difficult to accurately model ‘normal’ activity.

The idea of the anomaly analyser is try to obtain a measure of how unusual an alert is in relation to those previously logged. A further alert is triggered if this value is above a certain threshold. To determine this measure a weight is calculated for each alert based on the relative frequencies of individual alert features (i.e. source, destination, destination port, alert type) within the intrusion log. The alert weight,  $W_A$ , is calculated as follows:

$$W_A = \frac{1}{\sum W_F}$$

Where  $W_F$  is the relative frequency of each feature. The higher the  $W_A$  value the more unusual an alert is considered, the assumption being that if in the intrusion log there are a

large number of alerts containing certain features, such as a specific source or destination, then they can be assumed to be false positives, or at least that they have been explained and are considered to be of a low priority. It should, however, be reported when an alert that deviates from this normal activity is detected.

### 11.3 EXAMPLE INTRUSION DETECTION

To run the test, Snort and the snort sensor were started at each of the test sites. The analysers were started on the R-GMA server. NMap, a network mapping tool, was used to simulate the actions of an attacker attempting to scan the grid infrastructure by running a 'SYN stealth' scan against a node in each of the test sites.

Snort detected the port scan on all nodes scanned and alerts were created and logged, along with pseudo port scan packets of the following form:

```
[**] [122:1:0] (portscan) TCP Portscan [**]  
07/05-09:36:53.783140 134.226.53.60 -> 134.226.53.59  
PROTO255 TTL:0 TOS:0x0 ID:0 IpLen:20 DgmLen:163 DF
```

A Tcpdump sensor was run on a node being scanned, so that the traffic that triggered the alert could be collected with the following query:

```
SELECT sensorId, source_port, destination_port,  
       timestamp_Secs, timestamp_uSecs  
FROM TCP
```

Table 11.1 shows some of the packets collected by the sensor running on one of the targeted nodes during the scan. It shows a large number of TCP SYN packets (code 0x02) being directed at various ports on the target machine in a very short space of time. It also shows the response from the host machine to these packets. The majority of these responses

sensorId	source_port	destination_port	code	timestamp_Secs	timestamp_uSecs
gridmon.cs.tcd.ie:1	37456	5001	0x02	1120554368	256635
gridmon.cs.tcd.ie:1	5001	37456	0x14	1120554368	256672
gridmon.cs.tcd.ie:1	37456	574	0x02	1120554368	256773
gridmon.cs.tcd.ie:1	574	37456	0x14	1120554368	256786
gridmon.cs.tcd.ie:1	37456	839	0x02	1120554368	256790
gridmon.cs.tcd.ie:1	839	37456	0x14	1120554368	256802
gridmon.cs.tcd.ie:1	37457	1418	0x02	1120554368	569340
gridmon.cs.tcd.ie:1	1418	37457	0x14	1120554368	569375
gridmon.cs.tcd.ie:1	37457	4557	0x02	1120554368	569381
gridmon.cs.tcd.ie:1	4557	37457	0x14	1120554368	569393
gridmon.cs.tcd.ie:1	37457	280	0x02	1120554368	569398
gridmon.cs.tcd.ie:1	280	37457	0x14	1120554368	569408
gridmon.cs.tcd.ie:1	37457	2021	0x02	1120554368	569411
gridmon.cs.tcd.ie:1	2021	37457	0x14	1120554368	569422
gridmon.cs.tcd.ie:1	37457	498	0x02	1120554368	569425
gridmon.cs.tcd.ie:1	498	37457	0x14	1120554368	569435
gridmon.cs.tcd.ie:1	37457	26208	0x02	1120554368	569440
gridmon.cs.tcd.ie:1	26208	37457	0x14	1120554368	569450

**Table 11.1:** Sample of TCP traffic gathered by sensor during attack on TestGrid

will be RST packets (code 0x14), indicating to the source that no service is listening on this port. As described earlier it is this large number of negative responses that triggers the portscan alert. We can check for positive responses to the scan by querying for [SYN,ACK] packets (code 0x12). These will indicate to the attacker that a service is listening.

```
SELECT sensorId, source_port, destination_port, code,
timestamp_Secs, timestamp_uSecs
FROM TCP WHERE code = '0x12'
```

sensorId	source_port	destination_port	code	timestamp_Secs	timestamp_uSecs
gridmon.cs.tcd.ie:1	32771	37456	0x12	1120554370	829720
gridmon.cs.tcd.ie:1	111	37457	0x12	1120554371	149155
gridmon.cs.tcd.ie:1	22	37456	0x12	1120554374	823236

**Table 11.2:** Positive responses to port scan on TestGrid

Table 11.2 shows that there were in fact only three positive responses sent during the

port scan. Port 111 is the RPC portmapper process. It maps the RPC services available on a host to the ports they are using. In reality external access to this port should be blocked by a firewall as it can be used to provide a large amount of reconnaissance information to attackers. Port 32771 corresponds to another RPC service. Port 22 is the SSH daemon.

The portscan alert logged by each node is then sent to the QueryEngine, which in turn then streams the alert to the `snortAlerts` table. The alerts were collected at the R-GMA server by the Archiver (see Table 11.3). The various SNMP alerts seen in the log are triggered by the portscan sending packets to the default ports for various SNMP services. For example, the ‘SNMP AgentX/tcp request’ alert is caused by the attempted access to port 705, the default SNMP AgentX port. The ‘SNMP trap tcp’ alert is triggered when an attempt is made to access the default port of the SNMP Trap daemon, port 162.

```
SELECT siteId, sensorId, alert_timestamp, message, MeasurementTime
FROM snortAlerts
```

siteId	sensorId	alert_timestamp	message	MeasurementTime
giDCUie	gridmon.dcu.ie:0	07/05-09:41:57.075509	(portscan) TCP Portscan	08:42:08
giDCUie	gridmon.dcu.ie:0	07/05-09:41:56.755491	SNMP request tcp	08:42:07
giDCUie	gridmon.dcu.ie:0	07/05-09:41:21.876372	SNMP AgentX/tcp request	08:41:25
giDCUie	gridmon.dcu.ie:0	07/05-09:40:55.936382	(portscan) TCP Portscan	08:40:57
giDITie	gridmon.dit.ie:0	07/05-09:39:22.636994	(portscan) TCP Portscan	08:39:25
giDITie	gridmon.dit.ie:0	07/05-09:39:22.636988	SNMP request tcp	08:39:24
giDITie	gridmon.dit.ie:0	07/05-09:38:49.016974	SNMP AgentX/tcp request	08:38:53
giDITie	gridmon.dit.ie:0	07/05-09:38:47.097249	SNMP trap tcp	08:38:51
giDITie	gridmon.dit.ie:0	07/05-09:38:23.417710	(portscan) TCP Portscan	08:38:25
csTCDie	gridmon.cs.tcd.ie:0	07/05-09:36:53.783140	(portscan) TCP Portscan	08:37:25
csTCDie	gridmon.cs.tcd.ie:0	07/05-09:33:33.490563	SNMP AgentX/tcp request	08:34:04
csTCDie	gridmon.cs.tcd.ie:0	07/05-09:33:32.847380	(portscan) TCP Portscan	08:34:02
csTCDie	gridmon.cs.tcd.ie:0	07/05-09:33:33.163119	SNMP request tcp	08:34:03

**Table 11.3:** Sample of alerts logged to TestGrid grid-wide intrusion log

The port scan was detected by each of the example analysers. Each analyser generated its own ‘grid alert’ in response. Each grid alert is sent as an email, and also published to two

R-GMA tables, `gridAlerts` and `gridAlertTriggers`. These tables store details of the grid alert, and details of the snort alerts that triggered the grid alert respectively. *Archiving* these tables provides a persistent record of all grid alerts published, and also provides an index into the `snortAlerts` table so that the full alerts can be found.

The pattern matching analyser was set to generate an alert as soon as a portscan alert was detected at more than one site. Once detected the analyser generated an alert email of the following form:

Grid Alert: Grid Infrastructure Portscan

From:

<root@cagraidsvr17.cs.tcd.ie>

To:

stuart.kenny@cs.tcd.ie

Date:

05/07/2005 09:39:53

[\*\*] 07/05-09:39:53.946 Grid Infrastructure Portscan [\*\*]

Source: gridui.cs.tcd.ie (134.226.53.60)

Site: giDITie

07/05-09:39:22.636994 (portscan) TCP Portscan gridmon.dit.ie (147.252.15.28)

Site: giDCUie

07/05-09:39:50.802392 (portscan) TCP Portscan gridmon.dcu.ie (136.206.111.5)

Site: csTCDie

07/05-09:36:53.783140 (portscan) TCP Portscan gridmon.cs.tcd.ie (134.226.53.59)

For the heuristic analyser the threshold for alert inter-arrival time was set at 10,000ms, and the sample interval at 5 minutes. The portscan attempt, although not generating a large number of alerts, did produce alerts with an average inter-arrival rate below this value. As such a grid alert of the following form was generated:

Subject: Grid Alert: Alert Frequency Threshold Reached

From:

<root@cagraidsvr17.cs.tcd.ie>

To:

stuart.kenny@cs.tcd.ie

Date:

05/07/2005 09:39:47

[\*\*] 07/05-09:39:47.114 Alert Frequency Threshold Reached [\*\*]

Sample Interval: 07/05-09:34:46.710 -> 07/05-09:39:47.113

Average Alert Inter-Arrival Time (msecs): 9895

The anomaly analyser also detected the portscan. The initial portscan alert published to the log was assigned a weight of 0.98942834. The threshold for the test was set at 0.6. The weight was reasonably high as the source, gridui.cs.tcd.ie, and the alert type, had not frequently been seen previously in the log. The alert email generated was of the following form:

Grid Alert: Possible Malicious Activity

From:

<root@cagraidsvr17.cs.tcd.ie>

To:

stuart.kenny@cs.tcd.ie

Date:

05/07/2005 09:37:33

[\*\*] 07/05-09:37:33.139 Possible Malicious Activity [\*\*]

Source: gridui.cs.tcd.ie (134.226.53.60)

Alert weight: 0.98942834

Site (csTCDie): 0.012267511

Source (gridui.cs.tcd.ie , 134.226.53.60): 0.0043529877

Destination (gridmon.cs.tcd.ie , 134.226.53.59): 0.007518797

Port (-1): 0.9845667

Alert Type ((portscan) TCP Portscan): 0.001978631

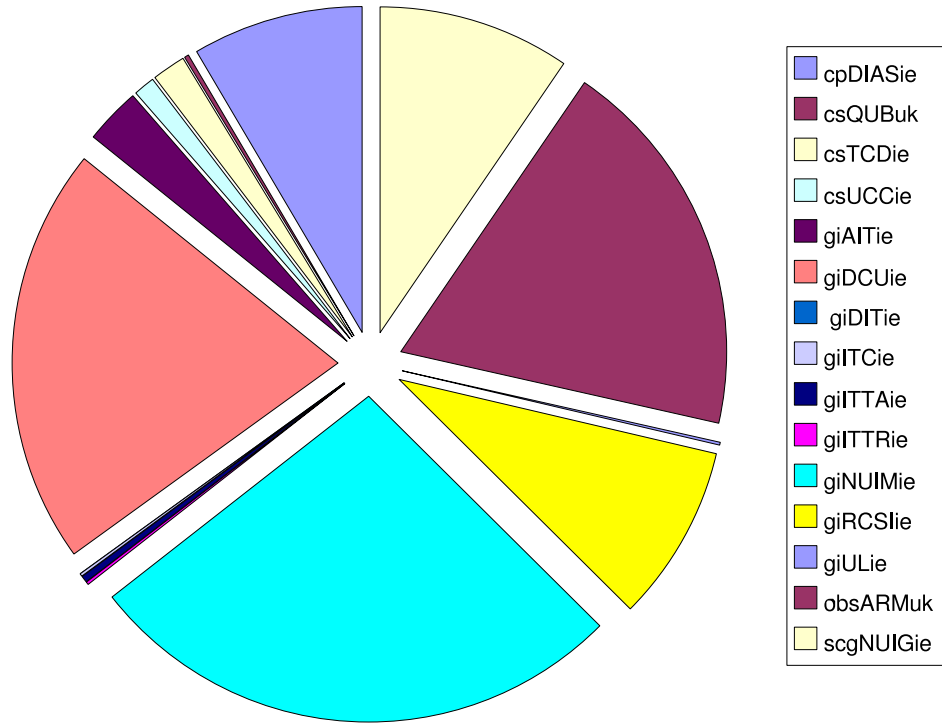
## 11.4 GRID-IRELAND DEPLOYMENT

An early prototype of GIDS has been deployed on Grid-Ireland. Snort, along with a Snort sensor, was started on a single node at 15 sites. Over a 4 week period 25,378 alerts were collected to the intrusion log by an archiver running on the central Grid-Ireland R-GMA server. Table 11.4 gives the distribution of alerts by site.

Site ID	No Of Alerts
cpDIASie	957
csQUBuk	25
csTCDie	194
csUCCie	124
giAITie	326
giDCUie	2352
giDITie	2
giITCie	13
giITTAie	41
giITTRie	13
giNUMie	17070
giRCSie	1012
giULie	17
obsARMuk	2157
scgNUIGie	1075

**Table 11.4:** Distribution of alerts by Grid-Ireland site

As can be seen there is significant variation in the number of alerts published. This is



**Figure 11.2:** Distribution of alerts by site, excluding the SNMP alerts at NUIM

mainly due to differences in firewall and network configurations and the services running on the networks within the sites. NUIM, for example, (National University of Ireland, Maynooth) accounted for approximately 70% of the total alerts, the majority of which ( $\sim 97\%$ ) were triggered by hosts within the NUIM network. 85% of these were SNMP alerts. All of the alerts logged for the site located in Armagh (obsARMuk) were generated by only two distinct hosts within the local network. Clearly these are examples of where the Snort rules must be tailored for the services, and configuration, of the site at which it is to be run (see Chapter 12). Figure 11.2 shows the distribution of alerts **excluding** the SNMP alerts generated by hosts within the NUIM network.

Table 11.6 shows the distribution of alert types seen over the 4 week period. As already

stated the majority of the alerts are SNMP related (due to the large number of alerts generated by hosts within NUIM). A significant number of alerts (approximately 17% of the total) related to Microsoft SQL Server (MS SQL) were also recorded. Table 11.5 shows an alert pattern that was often repeated in the intrusion log:

Time	Alert	Source	Destination
07/14-02:16:06.855078	MS-SQL Worm propagation attempt	61.185.142.14	136.206.111.7
07/14-02:16:06.855078	MS-SQL Worm propagation attempt OUTBOUND	61.185.142.14	136.206.111.7
07/14-02:16:06.855078	MS-SQL version overflow attempt	61.185.142.14	136.206.111.7

**Table 11.5:** MS SQL alert pattern

All of these alerts had a destination port of 1434, the MS SQL Monitor port, one of the top ten target ports as listed by DShield. This pattern is most likely caused by the SQL Slammer [37] worm, which targets a vulnerability in this service that allows for a buffer overflow exploit. As none of the nodes host MS SQL Server this is another example of how the Snort rules require tailoring to the grid environment. The remaining alerts are mostly various types of low priority scanning and reconnaissance alerts.

Alert Type	No Of Alerts
(http_inspect) BARE BYTE UNICODE ENCODING	156
(portscan) TCP Distributed Portscan	1
(portscan) TCP Portscan	91
(portscan) TCP Portsweep	27
(portscan) UDP Portscan	13
(portscan) UDP Portsweep	729
(snort_decoder) WARNING: TCP Data Offset is less than 5!	1
(spp_rpc_decode) Incomplete RPC segment	176
(spp_rpc_decode) Multiple RPC Records	1718
ATTACK-RESPONSES 403 Forbidden	6
ATTACK-RESPONSES id check returned userid	293

Alert Type	No Of Alerts
BAD-TRAFFIC tcp port 0 traffic	1
BAD-TRAFFIC udp port 0 traffic	21
DNS named version attempt	1
DNS SPOOF query response with TTL of 1 min. and no authority	301
ICMP Destination Unreachable	201
ICMP L3retriever Ping	313
ICMP Large ICMP Packet	26
ICMP PING CyberKit 2.2 Windows	181
ICMP PING NMAP	2039
ICMP redirect host	1
ICMP redirect net	1060
ICMP superscan echo	4
ICMP webtrends scanner	2
MS-SQL ping attempt	52
MS-SQL probe response overflow attempt	18
MS-SQL version overflow attempt	999
MS-SQL Worm propagation attempt	999
MS-SQL Worm propagation attempt OUTBOUND	999
RPC mountd UDP unmount request	1
RPC portmap listing TCP 111	1
RPC portmap mountd request UDP	2
RPC portmap proxy attempt UDP	70
RPC portmap rusers request UDP	49
RPC portmap ypserv request UDP	58
SCAN FIN	181
SCAN nmap XMAS	6
SCAN SSH Version map attempt	1
SNMP Broadcast request	126

Alert Type	No Of Alerts
SNMP broadcast trap	20
SNMP missing community string attempt	1
SNMP private access udp	24
SNMP public access udp	7074
SNMP request udp	7314
SNMP trap udp	20
WEB-MISC PCT Client_Hello overflow attempt	1

**Table 11.6:** Distribution of alert types

In the 4 week period a portscan of multiple Grid-Ireland sites from a single source was detected by the portscan analyser (described above). The following ‘grid-alert’ email was generated in response to this:

Grid Alert: Grid Infrastructure Portscan

From:

<root@cagraidsvr17.cs.tcd.ie>

To:

stuart.kenny@cs.tcd.ie

Date:

Yesterday 00:26:05

[\*\*] 08/04-00:26:05.244 Grid Infrastructure Portscan [\*\*]

Source: 59.44.51.80 (59.44.51.80)

Site: giULie

08/04-00:17:56.418485 (portscan) TCP Portscan gridmon.grid.ul.ie (193.1.96.134)

Site: giRCSIie

08/04-00:26:04.005235 (portscan) TCP Portscan gridmon.rcsi.ie (193.1.229.24)

Site: giAITie

08/04-00:13:41.395764 (portscan) TCP Portscan 192.168.32.154 (192.168.32.154)

As can be seen from the alert email the source IP, 59.44.51.80, scanned 3 hosts at 3 different sites, giULie (University of Limerick), giRCSIie (Royal College of Surgeons, Dublin) and giAITie (Institute of Technology, Athlone). The source had not appeared previously in the intrusion log, and no further activity was seen from it. The source IP is contained in the DShield (see Chapter 10) database however. At the time of writing 370 records, spanning a 4 day period, had been collected involving this host from monitored networks. According to the reports 5 ports were scanned: 80, 2301, 3128, 8000, and 8080. The purpose of the scan seems to be to identify active web proxies, as the host is scanning a set of ports frequently used by these (e.g. 3128 is the default port for the SQUID proxy server). Attackers frequently tunnel intrusions through vulnerable web proxies to hide their identity. As these scans were part of a larger scan there is no evidence of active targeting of Grid-Ireland sites, and this can be considered a false positive.

The ‘SCAN SSH Version map attempt’ alert published by the Snort sensor on the node located at the giITTAie (Institute of Technology, Tallaght) site did however warrant further investigation (see Figure 11.3). The reason this alert can be considered a higher priority is that it is targeting a service that is known to be running on the node being scanned. Vulnerabilities in the SSH daemon could lead to a node being compromised. Logs on the targeted node did show several attempted attacks on the SSH daemon, including the scan that triggered the Snort alert:

```
Jul 12 13:15:33 gridmon sshd[16120]: scanned from 128.252.74.67
with SSH-1.0-SSH_Version Mapper. Don't panic.
Jul 12 13:15:33 gridmon sshd[16119]: Did not receive
identification string from 128.252.74.67
```

The attacker attempted to determine the version of SSH running on the host, presumably to see if it was a version for which there are known exploits. Earlier in the log evidence of

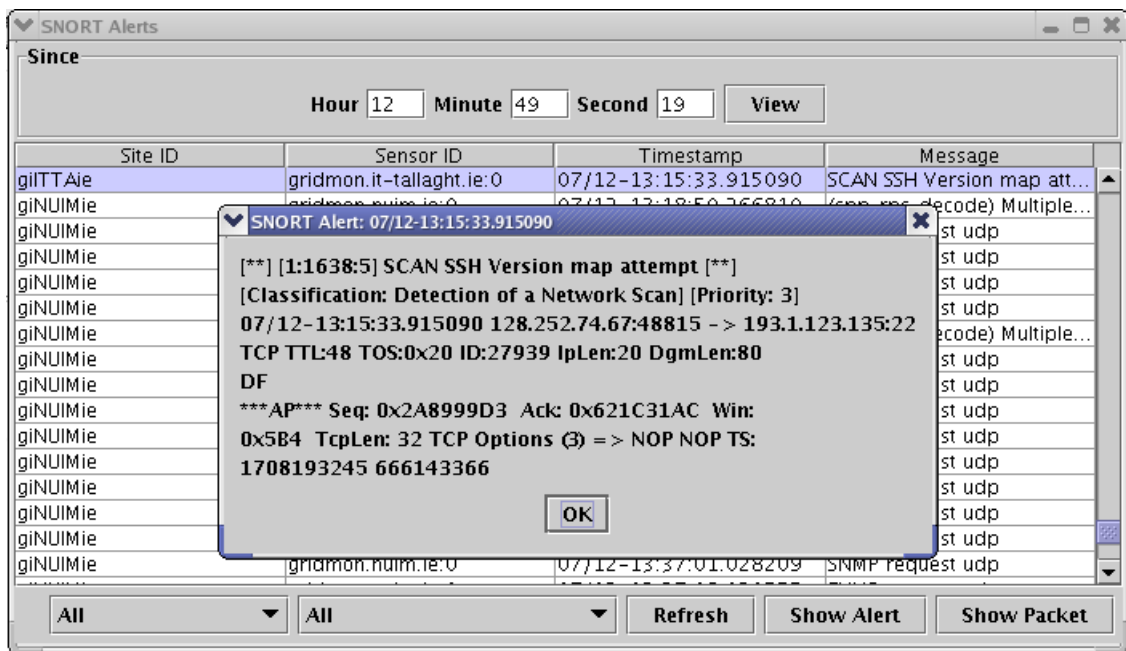


Figure 11.3: SSH alert viewed using Viewer GUI

further attacks from other source IPs could be seen. These were examples of ‘brute force’ attacks where random account names and passwords were used to try to gain access to the node:

```

Jul 12 11:53:50 gridmon sshd[15926]: Illegal user anna from 66.71.194.64
Jul 12 11:53:50 gridmon sshd(pam_unix)[15926]: check pass; user unknown
Jul 12 11:53:50 gridmon sshd(pam_unix)[15926]: authentication failure;
logname= uid=0 euid=0 tty=NODEVssh ruser= rhost=
h64.r194.clarion-limestone.iu6.usachoice.net
Jul 12 11:53:52 gridmon sshd[15926]: Failed password for illegal user anna
from 66.71.194.64 port 58518 ssh2Jul 12 11:53:54 gridmon sshd[15928]:
Illegal user ani from 66.71.194.64
Jul 12 11:53:54 gridmon sshd(pam_unix)[15928]: check pass; user unknown
Jul 12 11:53:54 gridmon sshd(pam_unix)[15928]: authentication failure;
  
```

```
logname=uid=0 euid=0 tty=NODEVssh ruser=  
rhost=h64.r194.clarion-limestone.iu6.usachoice.net  
Jul 12 11:53:56 gridmon sshd[15928]: Failed password for illegal user ani  
from 66.71.194.64 port 58921 ssh2  
Jul 12 11:53:58 gridmon sshd[15930]: Illegal user anca from 66.71.194.64  
Jul 12 11:53:58 gridmon sshd(pam_unix)[15930]: check pass; user unknown  
Jul 12 11:53:58 gridmon sshd(pam_unix)[15930]: authentication failure;  
logname=uid=0 euid=0 tty=NODEVssh ruser=rhost=  
h64.r194.clarion-limestone.iu6.usachoice.net  
Jul 12 11:54:00 gridmon sshd[15930]: Failed password for illegal user anca  
from 66.71.194.64 port 59305 ssh2  
Jul 12 11:54:01 gridmon sshd[15932]: Illegal user oana from 66.71.194.64  
Jul 12 11:54:01 gridmon sshd(pam_unix)[15932]: check pass; user unknown  
Jul 12 11:54:01 gridmon sshd(pam_unix)[15932]: authentication failure;  
logname=uid=0 euid=0 tty=NODEVssh ruser=rhost=  
h64.r194.clarion-limestone.iu6.usachoice.net
```

No successful logins were recorded. To prevent this type of attack access to the ssh port could be restricted to known IP ranges, such as just the local site and the Grid-Ireland OpsCentre.

Again, as for Chapter 9, such is the interest of these results that it is easy to lose sight of the fact that their acquisition is enabled by the SANTA-G framework, and that this chapter simply represents an example usage of this very generic RGMA-based framework.

## Chapter 12

# FUTURE WORK

There is significant scope for future work in expansion of the SANTA-G system. SANTA-G is a framework that is designed to allow a variety of sensor types, QueryEngines (i.e. instruments), and postprocessors (i.e. analysers).

### 12.1 SENSORS AND QUERYENGINES (INSTRUMENTS)

The NetTracer described in this thesis is a demonstrator of both a SANTA-G software (Tcpdump and Snort) and hardware (SCI trace hardware) instrument. With regard to the ethernet NetTracer, the schema, although quite complete, can be extended quite easily. There is data logged by Tcpdump that is not yet utilized, for example the options fields of the IP, TCP and UDP headers. An essential future extension is to provide support for IPv6 packets. Currently only IPv4 is supported. There have as yet only been some initial simple attempts at improving the search mechanism of the QueryEngine. Currently a linear search of the log files is used to retrieve the data. There are a number of classical search algorithms that could be applied to improve the performance. Grover's algorithm [14], for example, is a quantum algorithm that would take  $O(N^{1/2})$  to search an unsorted database containing  $N$

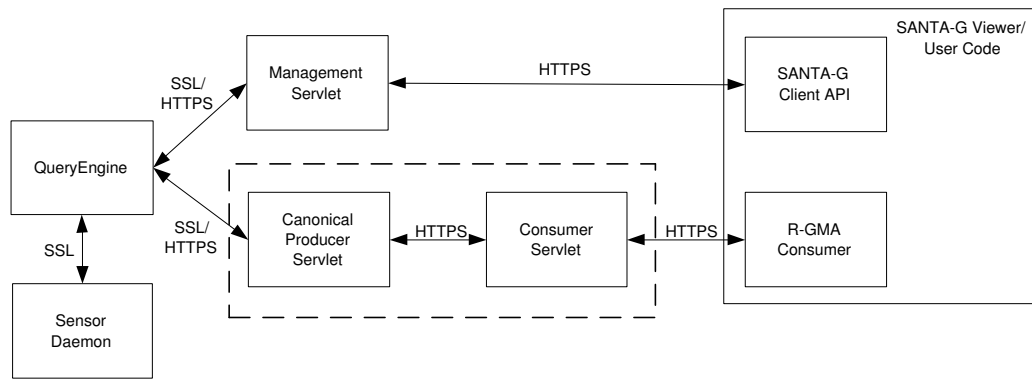
entries, compared to the  $O(N)$  time taken by a linear search.

The version of the R-GMA used during development of the NetTracer, and described in this thesis, is the version included in the LCG [28] 2.3 grid middleware release, upon which the final release of the CrossGrid middleware is based. As part of the refactoring effort taking place within the EGEE project there have been several subsequent LCG and R-GMA releases. Major changes include moving from a servlet based system to web-services, and the introduction of a new API. The various producer types, Stream, Latest and DataBase, are now referred to as Primary Producers, with Archivers becoming Secondary Producers, as they republish the information published by a Primary. The Canonical Producer is now the OnDemand Producer. Producers are created by producer factories, with producer properties specifying the required behaviour of the producer to create. An essential future task is the migration of the NetTracer to the new API. This will affect both the QueryEngine and Viewer components. The majority of the changes will be minor, involving a change in the API calls. Certain operations will require further effort, such as the method used to remove closed sensors from the sensor information tables (see Section 7.1.2). In the new API the concept of ‘clean-up predicates’ that specify tuples to remove, has been replaced by ‘retention periods’ that specify the amount of time for which a producer should retain a tuple. To allow for this the QueryEngine will need to republish the information for the currently running sensors at regular intervals less than the retention period. Any tuple older than the retention period can be assumed to be for a sensor that has since been closed and can therefore be removed.

A major issue yet to be addressed is that of security. The NetTracer badly needs a method for authorisation of users attempting to retrieve data from the system. This would have to be based on the standard grid security mechanisms such as GSI using certificates. Because the system publishes network data it is important that only users that should be allowed view the data are given access. The R-GMA does provide a simple authentication mechanism that requires a user to have a valid grid certificate in order to connect to the Consumer Servlet. This does not, however, provide the means to decide whether a user should be allowed access

to particular tables of data once they have connected. Future releases of the R-GMA will provide for the authorisation of users. Another requirement is for the remote management of sensors. This is important for scalability. This feature relies on authorisation being present, as only users with the correct access privileges should be allowed to control sensors.

A secure NetTracer prototype that includes some of these features has been developed. A management module has been added that provides a simple authorisation scheme and also remote management of sensors. The module is comprised of a management servlet, a sensor daemon, and a client API. Figure 12.1 shows the structure of the prototype.



**Figure 12.1:** Secure NetTracer prototype structure

The ManagementServlet is used to authorise users, and to inform the QueryEngine when a new user has been successfully authorised. Authorisation is performed by an authorisation filter, developed by the EDG project as part of its Java security package. The client API allows a user to interact with the ManagementServlet. When sending an authorisation request, a key is generated for the user by the client API. If authorisation is successful the key is accepted and forwarded to the QueryEngine. The key is then included in the WHERE predicate of submitted queries, e.g.:

```
SELECT * FROM Ethernet
WHERE authorised_key=
```

```
’/C=IE/O=Grid-Ireland/OU=cs.tcd.ie/L=RA-TCD/CN=Stuart P. Kenny  
2402e21f621397f6beb1cdef317bad6b54ca744’
```

Results are only returned for queries containing keys known to the QueryEngine. The API and servlet also provide methods for both starting and stopping sensors on remote hosts. Requests to control sensors, from users with the correct access privileges, are routed by the servlet through the QueryEngine to the correct sensor daemon. The sensor daemon invokes the sensor on the host machine using the configuration contained in the request. It is hoped to include some of these features in the deployed NetTracer release in the near future.

The QueryEngine examples described in this thesis support libpcap format log files and also the SCI trace files created by the original SANTA tools. It is possible to create new sensor types and QueryEngine implementations to support other instruments and log file formats. The grid contains many possible sources of information. Grid services, such as the Globus Gatekeeper and the PBS queue manager, produce large volumes of log file data. Although this allows for detailed tracking and analysis of resource usage, finding the relevant information amongst the verbose output can be difficult. This is an area where the SANTA-G approach would be beneficial. A SANTA-G implementation capable of searching system log files such as these would provide a powerful means of debugging the grid middleware and of tracing individual job submissions. Complementing the information published in libpcap format by Snort with that obtained from other security components would also be useful. Snort is a network intrusion detection system, which means it is incapable of detecting activity that takes place on the host machine itself, such as unauthorised access of system files. Providing additional querying of the logs of a host-based system, such as Tripwire, would allow for the detection of both attempted intrusions (by Snort) and the tracking of successful intrusions (by Tripwire). This is the topic of an impending Msc thesis. It is also intended to investigate other IDS such as Bro [36], a ‘Globus-aware’ NIDS. This system is capable of tracking GSI connections and so could be used to verify and track ‘legitimate’ resource usage. With regard

to additional instruments the NetTracer can be extended for other interconnect technologies by making use of existing hardware trace instruments. IBTracer [40], for example, is a protocol analyser capable of tracing traffic from an *InfiniBand<sup>TM</sup>* link.

## 12.2 POSTPROCESSORS (ANALYSERS)

Perhaps the most basic of the SANTA-G postprocessors is that which filters NetTracer logs for one-way transit times, as described in Chapter 9. The one-way transit time is, however, one of many simple yet very desirable metrics that otherwise may only be obtained through the Test Traffic Measurements (TTM) [21] service offered by RIPE NCC [41], a service organisation for Internet Service Providers. Availing of this service involves purchasing a RIPE ‘test box’. The transit time metric may be used to optimise file transfers, messaging behaviour, and even distributed shared memory systems. Grid-Ireland intend to use it to log an audit trail of site time offsets, and to use this for auditing of security intrusion detection timestamps.

The grid-wide intrusion detection system described in this thesis is a demonstrator of a SANTA-G analyser of NetTracer logs. The issue of grid security is becoming an increasingly important consideration in recent grid projects. As a large scale distributed computing infrastructure a grid will be vulnerable to the same threats as traditional networks, at both network and individual host level. There are also some grid specific issues. In order to provide access to sites across organisations trust relationships must exist between hosts. These relationships can be exploited, particularly in the case of stolen user credentials, which would give an attacker access to sites that would be extremely difficult to trace as it would appear as legitimate usage. The recent shift in grid middleware to a web-services based infrastructure also brings new security threats. There are several well known exploits for web-services, all of which the grid will inherit, such as WSDL scanning, SQL injection, and Replay Attacks [23]. Also because most sites will tend to have similar infrastructures in terms of operating systems and services, a successful attack at one site could very quickly be

repeated at another [27]. All of this will lead to the need for stringent security monitoring, with the ability for fine-grained analysis of inter-site security incidents. It is expected that the work started in this thesis on the grid-wide intrusion detection system will play a role in this, and that the majority of my future work in the near future will be in this area.

The initial steps for this work has begun with the deployment of an early prototype of GIDS on Grid-Ireland. It will be useful to run the system over a period of time in order that a baseline of ‘normal’ activity can be gathered. For example certain alerts can be triggered by everyday grid tasks such as updating the RPMs on hosts. The use of NFS by LCFG for this triggers several RPC Snort alerts. False alarms such as these will need to be understood and filtered in order that malicious activity can be detected amongst the alert ‘noise’. Alerts that fall outside the expected patterns could be considered as being part of an attack.

Once a baseline model has been obtained it could be used to develop a custom set of Snort rules specifically tailored to a grid. Rulesets for host types, such as Computing Elements and User Interfaces, could be developed for exploits particular to the services found on those hosts. This work would have the benefit of significantly reducing the rates of ‘false positives’ and therefore increasing the chances of detecting actual attempted exploits. As well as developing rule sets it may be necessary, and beneficial, to develop new Snort preprocessors, again tailored to detecting exploits of known, or predicted, grid vulnerabilities.

The intention of the work described above is to improve the detection of alerts at the host/site level, in terms of reducing the false positive rate by tailoring the alerts more to the grid environment. This needs to be done in order to reduce the load on the analyst/analysis software that is examining the detected alerts. In the case of GIDS this analysis will be carried out by custom code that uses R-GMA to filter the grid-wide intrusion log to detect patterns signifying attempted attacks. The development of this code will form the bulk of my effort in the near future. Again the initial steps will involve a detailed analysis of the expected attacks that could be leveraged against a grid infrastructure. Some work has already been undertaken in this area by the EGEE project (see [17]), however it will not be until grids

become more prevalent, and security incidents more frequent, that grid exploits will become more widely known. Once the patterns, or signatures, of these attacks have been determined, then the code to detect them can be developed.

Clearly detection is a pattern matching process, with the probability of false positives and negatives depending on the sample size and the autocorrelation function of the pattern, amongst other things. The creation of a grid-wide log increases the sample size and therefore should increase the confidence level for the resulting alerts. The pattern matching process is very likely to benefit from a statistical approach such as Bayesian filtering. Bayes' formula allows for the expression of the probability of an event as a combination of the probabilities of other independent events. This approach is frequently used in the identification of spam email. In order to do this a filter must first be trained. In the case of spam detection training is carried out on two sets of email, one known to be spam and one known to be legitimate. This approach could be applied to the classification of alerts contained in the grid-wide log. Training sets could be created by adding alerts from the intrusion log that are known to be malicious, or that are unexplained, to an incident database. This incident database forms the 'malicious activity' set, whereas any alert contained in the intrusion log, but not in the incident database, can be considered a member of the 'false positive' set. A Bayesian filter could then be trained using this data to classify alerts, based on features (e.g. source, destination, alert type) contained in the alert, in a similar way as words contained in an email are used for spam identification. The probability of an alert being malicious, given the features in the alert, can be expressed using Bayes' formula as follows:

$$P(\text{malicious}|\text{features}) = \frac{P(\text{features}|\text{malicious})P(\text{malicious})}{P(\text{features})}$$

If it is determined that an alert has a high probability of being malicious then a further 'grid alert' can be triggered. This approach could be used in the relatively simplistic case of classifying single alerts as either possibly malicious or false positive. For more detailed analyses, such as recognising patterns of activity across sets of alerts contained in the log, more complex pattern classification systems, possibly using AI techniques and category theory,

would be required.

Although in initial implementations it is expected that upon detection of an attack the response will be to generate and deliver further alerts, another area of future work will be in developing more ‘active’ responses to threats. Although it is hoped that the time between attempted attack and attack detection/notification will be significantly reduced by the GIDS described here, stopping the attack will still depend on the response time of the person alerted. Alerts may not be checked, or may be delayed if systems such as email or SMS are relied upon. To preclude this an automated response will be attempted. For example in the simple port scan case described in the previous chapter the source IP could be added to a firewall block list, and immediately denied access through the firewall. Care would have to be taken with this type of approach, however, to ensure that it was not itself exploited. In the above, spoofing an IP address would lead to an innocent party being denied access to services, effectively a DoS attack on the spoofed system.

In terms of scaling, it seems sensible that there be national GIDSs, as for Grid-Ireland, that publish derived alerts in an international hierarchy such as the federated hierarchy of EGEE.

## Chapter 13

# CONCLUSION

### 13.1 OVERVIEW

The objective of this research was to provide ‘a framework to allow for ad-hoc monitoring experiments in the grid environment’, along with the implementation of a set of tools that would form a demonstrator of this concept. This has been achieved.

This research has contributed to two major EU grid projects, DataGrid and CrossGrid. The central component of the framework, the Canonical Producer, is now an integral part of the DataGrid R-GMA. The NetTracer system was developed within the CrossGrid project as one component of its grid monitoring system. As well as these projects the NetTracer system is also being used by Grid-Ireland, the national computational grid of Ireland, as a network monitoring tool and as the basis of an intrusion detection system.

### 13.2 PERSONAL SUMMARY

Overall the research has been a success. It has achieved the goals set out at the beginning. As stated the results of the research have been exploited by three major grid projects. I have learnt a great deal from my involvement in these projects, as well as in the process of de-

signing and implementing the SANTA-G NetTracer system. This includes new technologies such as R-GMA and LCFGng, and also project management tasks, such as writing Software Requirement Specifications, Software Design Documents, and providing detailed user documentation, installation and developer guides. It has also allowed me to learn how to move from academic notions, to a proof of concept, to approximately 16,000 lines of production quality code. The necessity to provide system and unit tests and to adhere to strict developer guidelines and testing and validation procedures has been very beneficial. It seems likely too that conception of the grid-wide intrusion detection has opened up a whole new, if specialised, research area, and this is very satisfying.

# Bibliography

- [1] IEEE 1596, *IEEE standard for scalable coherent interface, IEEE std 1596-1992*, IEEE Computer Society, August 1993.
- [2] Fingerprinting Sharing Alliance, <http://www.arbornetworks.com/fingerprint-sharing-alliance.php>, April 2005.
- [3] Sergio Andreozzi, Natascia De Bortoli, Sergio Fatinel, Antonia Ghiselli, Gennaro Tortone, and Cristina Vistoli, *GridICE: a monitoring service for the Grid*, Proc. 3<sup>rd</sup> Grid Workshop, Cracow, October 2003.
- [4] Ruth Aydt, Dan Gunter, Warren Smith, Martin Swany, Valerie Taylor, Brian Tierney, and Rich Wolski, *A grid monitoring architecture*, (2001).
- [5] Jay Beale, James C. Foster, Jeffrey Posluns, and Brian Caswell, *Snort 2.0 intrusion detection*, Syngress Publishing Inc., 2003.
- [6] Rob Byrom, Brian Coghlan, Andrew Cooke, Roney Cordensoni, Linda Cornwall, Ari Datta, Abdeslem Djaoui, Laurence Field, Steve Fisher, Stuart Kenny, James Magowan, Werner Nutt, David O'Callaghan, Manfred Oevers, Norbert Podhorski, John Ryan, Manish Soni, Paul Taylor, Antony Wilson, and Xiaomei Zhu, *R-GMA: A relational grid information and monitoring system*, Proc. 2nd Cracow Grid Workshop, December 2002.

- [7] Rob Byrom, Brian Coghlan, Andrew Cooke, Roney Cordenonsi, Linda Cornwall, Ari Datta, Abdeslem Djaoui, Laurence Field, Steve Fisher, Stuart Kenny, James Magowan, Werner Nutt, Manfred Oevers, David O'Callaghan, Norbert Podhorski, John Ryan, Manish Soni, Paul Taylor, Antony Wilson, and Xiaomei Zhu, *The CanonicalProducer: an instrument monitoring component of the Relational Grid Monitoring Architecture*, Proc. 3rd ISPDC, IEEE Computer Society, July 2004, pp. 232–237.
- [8] Brian Coghlan, Abdeslem Djaoui, Steve Fisher, James Magowan, and Manfred Oevers, *Time, information services and the Grid*, BNCOD 2001 - Advances in Database Systems (K D Oneill and B J Read, eds.), RAL-CONF, no. RAL-CONF-2001-003, BNCOD, 2001.
- [9] A. Cooke, A. Gray, L. Ma, W. Nutt, J. Magowan, P. Taylor, R. Byrom, L. Field, S. Hicks, J. Leake, M. Soni, A. Wilson, R. Cordenonsi, L. Cornwall, A. Djaoui, S.M. Fisher, N. Podhorszki, B. Coghlan, S. Kenny, and D. O'Callaghan, *R-GMA: An information integration system for grid monitoring*, Proc. of the Tenth International Conference on Cooperative Information Systems, 2003.
- [10] Nathan Einwechter, *An introduction to distributed intrusion detection systems*, InFocus Security Focus Article, <http://www.securityfocus.com/infocus/1532> (2001).
- [11] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke, *A directory service for configuring high-performance distributed computations*, Proc. 6th IEEE Symp. on High Performance Distributed Computing, IEEE Computer Society Press, 1997, pp. 365–375.
- [12] Vangelis Floros, Christos Markou, and Nikos Mastroyiannopolous, *Cluster installation in CrossGrid LCFGng for dummies (v1.0b2)*, [http://cgi.di.uoa.gr/~xgrid/cgfiles/LCFGng\\_v1.0.pdf](http://cgi.di.uoa.gr/~xgrid/cgfiles/LCFGng_v1.0.pdf), January 2004.

- [13] I. Foster and C. Kesselman, *The Grid: Blueprint for a future computing infrastructure*, Morgan Kaufmann Publishers, 1998.
- [14] L. K. Grover, *A fast quantum mechanical algorithm for database search*, Proc. 28th Annual ACM Symposium on the Theory of Computing, May 1996, p. 212.
- [15] Paul Innella and Oba McMillan, *An introduction to intrusion detection systems*, InFocus Security Focus Article, <http://www.securityfocus.com/infocus/1520> (2001).
- [16] Steve J. Chapin, Dimitrios Katramatos, John Karpovich, and Andrew Grimshaw, *Resource management in Legion*, Future Generation Computer Systems **15** (1999), no. 5-6, 583–584.
- [17] EGEE JRA3, *Grid security incident description and exchange format*, <https://edms.cern.ch/file/501422/1.2/EGEE-JRA3-TEC-501422-Grid-Security-Incident-v-1.2.pdf>, October 2005.
- [18] Stuart Kenny, *Statistical analysis of non-invasive scalable coherent interface trace data*, final year project report, Computer Science Department, Trinity College Dublin, June 2001.
- [19] Stuart Kenny and B. A. Coghlan, *Towards a Grid-wide intrusion detection system*, Proc. European Grid Conference, LNCS 3470 (Amsterdam, The Netherlands), February 2005, pp. 275–285.
- [20] Stuart Kenny and Brian Coghlan, *Grid-wide intrusion detection*, Proc. 3rd Cracow Grid Workshop (Cracow, Poland), December 2004, pp. 331–337.
- [21] Olaf Kolkman and Henk Uijterwaal, *Internet delay measurements using test traffic*, <http://www.ripe.net/test-traffic>, May 1997.

- [22] Fang-Yie Leu, Jia-Chun Lin, Ming-Chang Li, Chao-Tung Yang, and Po-Chi Shih, *Integrating grid with intrusion detection*, Proc.AINA 2005 (Taipei, Taiwan), vol. 1, March 2005, pp. 304–309.
- [23] Pete Lindstrom, *Attacking and defending web services*, Spire Research Report, [http://forumsystems.com/papers/Attacking\\_and\\_Defending\\_WS.pdf](http://forumsystems.com/papers/Attacking_and_Defending_WS.pdf) (2004).
- [24] Manzke M. and Coghlan B.A, *Non-intrusive deep tracing of SCI interconnect traffic*, Proc. SCIEurope'99, no. ISBN82-14-00014-9, SINTEF Electronics and Cybernetics, September 1999, pp. 53–58.
- [25] M. Manzke, S. Kenny, B. Coghlan, and O. Lysne, *Tuning and verification of simulation models for high speed interconnect fabrics*, Proc. PDPTA 2001, June 2001.
- [26] Matthew L. Massie, Brent N. Chun, and David E. Culler, *The Ganglia distributed monitoring system: Design, implementation and experience*, Parallel Computing **30** (2004), no. 5-6, 817–840.
- [27] Andrew McNab, *Security monitoring boxes*, <http://agenda.cern.ch/askArchive.php?base=agenda&categ=a053292&id=a053292s1t10/transparencies>, GridPP Deployment Board Meeting, Glasgow, June 2005.
- [28] LCG Grid Middleware, <http://lcg.web.cern.ch/lcg/activities/middleware.html>, July 2005.
- [29] Stephen Northcutt, Mark Cooper, Matt Fearnow, and Karen Frederick, *Intrusion signatures and analysis*, New Riders Publishing, 2001.
- [30] Stephen Northcutt and Judy Novak, *Network intrusion detection, 3rd edition*, New Riders Publishing, 2002.
- [31] G. Pfister, *In search of clusters, 2nd edition*, Prentice Hall PTR, NJ, 1998.

- [32] Gunnar Ronneberg and Olav Lysne, *An Opnet-based simulation model of SCI-nodes*, Proc. SCI Europe 1999, no. ISBN82-14-00014-9, SINTEF Electronics and Cybernetics, 1999, pp. 101–112.
- [33] Dolphin Interconnect Solutions, *Link Controller 3 specification D666 - LC-3*, <http://www.dolphinics.com/products/hardware/lc3.html>, v1.9 ed., June 2002.
- [34] Cisco Systems, *The science of intrusion detection system attack identification, white paper*, [http://www.cisco.com/warp/public/cc/pd/sqsw/sqidsz/prodlit/idssa\\_wp.pdf](http://www.cisco.com/warp/public/cc/pd/sqsw/sqidsz/prodlit/idssa_wp.pdf).
- [35] Brian Tierney and Dan Gunter, *NetLogger: A toolkit for distributed system performance tuning and debugging*, Tech. Report LBNL-51276, Lawrence Berkeley National Laboratory, 2002.
- [36] Bro Website, <http://www.bro-ids.org>, June 2005.
- [37] CERT Website, <http://www.cert.org/advisories/ca-2003-04.html>, July 2005.
- [38] Chromium Website, <http://chromium.sourceforge.net>, August 2005.
- [39] DShield website, <http://www3.niu.edu/mpi/>, April 2005.
- [40] LeCroy Website, <http://www.lecroy.com/tm/products/protocolanalyzers/ib.asp?menuid=62>, July 2005.
- [41] RIPE NCC Website, <http://www.ripe.net/info/ncc/index.html>, July 2005.
- [42] Snort Website, <http://www.snort.org>, April 2005.
- [43] The EU CrossGrid Project website, <http://www.eu-crossgrid.org>, June 2005.
- [44] Rich Wolski, Neil T. Spring, and Jim Hayes, *The Network Weather Service: A distributed resource performance forecasting service for Metacomputing*, Future Generation Computer Systems **15** (1998), no. 5-6, 757–768.

- [45] DataGrid WP2, *EDG java security*, <http://edg-wp2.web.cern.ch/edg-wp2/security/edg-java-security.html>, September 2005.