Visualising and Scanning Network Information

Author
Leon Roy

14th June 2006

Supervisor: Dr. Emil C. Lupu
Second Marker Prof. Morris Sloman

Table of Contents

Table of Figures

Abstract

The proliferation of computer networks with features previously only found in corporations and academic institutions has led to more types of traffic taking advantage of cheap, scalable networks. High speed devices with multiple functions that used to cost thousands of pounds for a single function alone are becoming more common. But with an increase in services being offered to network users comes an increase in complexity. Networks are becoming more time consuming and difficult to administer as the amount of information within them has grown.

Current visualisation tools offer complex interfaces and rely heavily on user input to produce representative network topology diagrams. Display such topologies as two-dimensional diagrams they are limited in the amount of data they can show. Data traffic between network appliances is shown in charts on a separate window when in three-dimensions it can be overlaid on the diagram itself.

What is needed is a means of visualising networks in a convenient way so as to facilitate their management, and to identify and troubleshoot particular problems. The aim of this project is to design a tool which is able to represent networks in a three-dimensional way and to display network information as a part of the diagram itself.

Acknowledgements

I would like to express my thanks and gratitude to:

- My supervisor, Dr. Emil Lupu, for supervising my project and for his help and guidance regarding it.
- My second marker, Professor Morris Sloman, for his encouragement and friendliness.
- My father, for his patience and persistence when I used him as a sounding board for some of my crazier ideas.

1 Introduction

Network traffic information is difficult to visualise effectively. The flexibility of networks and their dynamic nature has meant that the information required to represent them is never static. Increased demand for accessibility to the network across wireless, public and private networks (eg. VPN), and the addition of services and features to network appliances has made them more complex to troubleshoot. The increasing number of application layer protocols transmitted across a network and the variety of appliances communicating has increased the amount of information required to represent the network's security at any point in time.

With the variety of information going across a network, focusing on a pertinent portion of it becomes a daunting task with such a profusion of data to sort through. Network tools that aid diagnosis of the network can either overwhelm the user with too much irrelevant data or provide them with too simple a view of the network. Representation of contextual information; the network interface, the service being used and whether the data is encrypted or not add increasing amounts of information that make visualisation more complicated. The problem with this being not only too much information being presented, but adequately displaying all of it without confusing the viewer. It would be desirable to have a means of visualising the network in an intuitive manner with an abstract view at the top most level to aid navigation to a pertinent point and then increasingly lower level views showing more detailed information. This would allow the user to focus on a particular security aspect of their network or a particular vulnerability without being distracted by unnecessary information.

The objectives of this project are to use data visualisation techniques to display network security information graphically, provide users with a tool that accurately represents the network from the physical layer to progressively more abstract layers, to provide a representation that is both easily navigable and flexible in the level of information it can show and to provide an effective network security diagnostic tool.

The challenges faced are:

- Network information is dynamic in nature, like hitting a moving target, the information being displayed often has a delay associated with it.
- Network traffic has context, eg. The network layer it travelled over, the network port it is intended for, whether it is encrypted or not, and whether the destination server has any security issues the user should be aware of.
- Representing a network's full state 100% accurately at any given time requires every single piece of information in the network.
- Representing a network as accurately as possible using only a subset of that information

1.1 Objectives

The objectives for this project are as follows:

- Research visualisation techniques for representing network topologies, traffic across networks and the representation of individual network appliances in three-dimensions.
- Settle upon the design of the network visualisation tools, the techniques required to support them and the network appliances required to test them.
- Create a user interface that is sufficiently intuitive that it allows first time users to understand it with minimal instruction, but also allows advanced users to see as much information as possible.
- Test the visualisation tool, its reliability, accuracy of representation and utilising user reporting make possible refinements.

Scanning and representing network topologies is difficult to do because the information required to display a network straddles protocols and network scanning programs as diverse as Simple Network Management Protocol (SNMP), Nmap, Nessus and Syslog to name a few. Furthermore, representing information from so many sources requires dealing with the inherent latencies and the changing state of a network. The program therefore must be well designed and robust, able to deal with failed network requests and network time outs. It must be able to provide an appealing and easy to use graphical-user-interface without such time outs causing the user-interface to hang the application.

1.2 Requirements

Data Collection

- The system will need to integrate information from a number of sources. These include SNMP, Nmap and Nessus. Supporting these data sources will require custom libraries as well integrating those written by external parties.
- The system will need to be able to interrogate such sources across the network and as such will need to be able to handle network timeouts and latency as well as poll such devices to update its view.

Unified Network Model Representation

- A network composed of devices such as switches, routers and computers is complex. It consists not only of these elements but their network interfaces and services as well. To construct a detailed representation of a network a unified data structure will need to be used.

- The structure will need to be flexible and extensible; able to deal with many types of network elements.
- It will need to be able to hold information from a variety of sources so that a full picture of the network can be produced.
- The network model will ideally conform with an industry agreed standard such as Simple Network Management Protocol or the Common Information Model.
- The model should be interoperable with additional data sources so that they can be integrated at a later date.

User Interface

- The user interface should be intuitive and obvious to the user and allow them to begin effective interaction easily.
- The user interface should assist the user with any new concepts in a non-obtrusive manner.
- The user interface should be flexible in the level of information it presents the user, meeting effectively the user's information criteria.
- The user interface should be flexible in the ways in which user and interface can exchange information.

Data Visualisation

- The system should provide the user as complete and accurate a visual representation of the network topology as possible.
- The system needs to be able to provide as close to real-time visual data analysis of the network's current state as possible.
- Network traffic information as well as the status of individual network devices should be integrated with the network topology diagram wherever possible.

Interoperation

- The system should be cross platform, maximising its scope for use as a network tool from as many operating systems as possible.
- The system should be scalable to cope with multiple network agents and multiple clients, using standardised protocols wherever possible to maximise compatibility with network devices.

1.3 Report Structure

- Chapter two examines related work and explores ideas for visualising networks and their information.
- Chapter three outlines the design of the system, the functionality planned and any problems foreseen.

- Chapter four details how the system is implemented as well as describing technical decisions and hurdles that had to be overcome in the process.
- Chapter five evaluates the network visualisation tool and the strengths and weaknesses of its approach as well as detailing the testing of the system
- Chapter six concludes the project, detailing the challenges faced and future work.

2 Background

This chapter evaluates related work and explores ideas for implementation

2.1 Visualisation

This chapter details related and relevant work, beginning with a discussion of each item as well as its advantages and disadvantages.

DNA Sequence Walkers

Representing DNA sequences visually, 'DNA Sequence Walkers'[1] use shape, colour and orientation to show how DNA sequences and other macro molecules interact with other molecules.

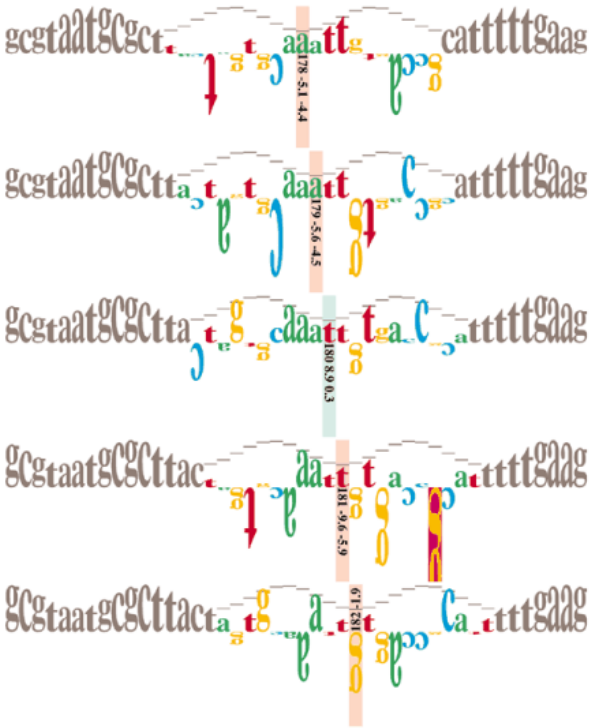

Figure 1 – Diagram showing DNA sequence walker progressing along the sequence

The walker is shown by the coloured letters and the cosine wave the orientation of the DNA facing a molecule. Characters representing the sequence being analysed are either oriented upright above the line, representing favourable interaction, or are oriented below the line upside-down, representing unfavourable contact. The height of the letters (bases) form a 'wave', representing the helix shape of the sequence with individual heights within the analysed section representing the information content of that base in bits.

The DNA Sequence Walkers method represents at least five different pieces of

information visually. It uses height, orientation, colour, shape and because it occurs over a period of time, movement, to represent different characteristics of a DNA sequence.

When representing information visually, it is important to utilise the viewer's expectations of the object being represented. The walker meets the user's expectation of a three-dimensional DNA helix representing it as a two-dimensional cosine wave. In addition it represents the strength of each base using height, and a positive binding in the positive Y direction (above the line) and a negative binding in the negative Y direction, again conforming to convention, and so the user's expectations making the system easier to read since the way the information being conveyed is already familiar to the user: I would call using the user's pre-conceived notions as the system being 'intuitive'.

MRTG Type Graphs

Traffic graphers such as MRTG use graphs which represent network traffic visually. Like DNA Sequence Walkers they represent data over time, with colour, height and shape being used to show network traffic visually.

In MRTG, network traffic on a graph displays the amount of data in bits per second on the positive Y-axis, with time on the positive X-axis. Network traffic outbound is represented as a line in one colour, while network traffic inbound is represented as a shaded area in a different colour.

Again, size represents quantity the Y-axis quantifying the amount of network traffic and time conveyed by movement along the X-axis. This is obvious when representing data on a graph and accordingly the graph format conforms to the user's expectation.

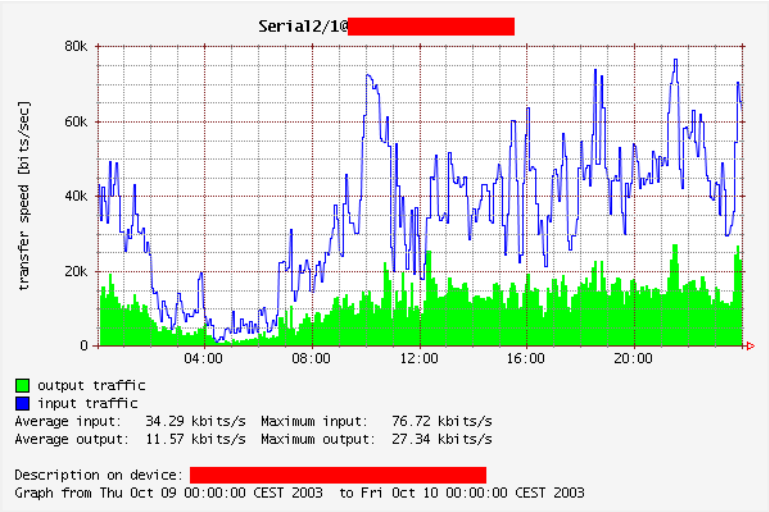

Figure 2 – MRTG Graph of network traffic inbound and outbound

Visualisation tools for network information such as FlowScan and RRD take this one step further, representing outflows on the positive Y-axis and data inflows on the negative Y-axis, with shaded areas representing the different types of network traffic (eg. WWW, FTP, SMTP, TCP-other).

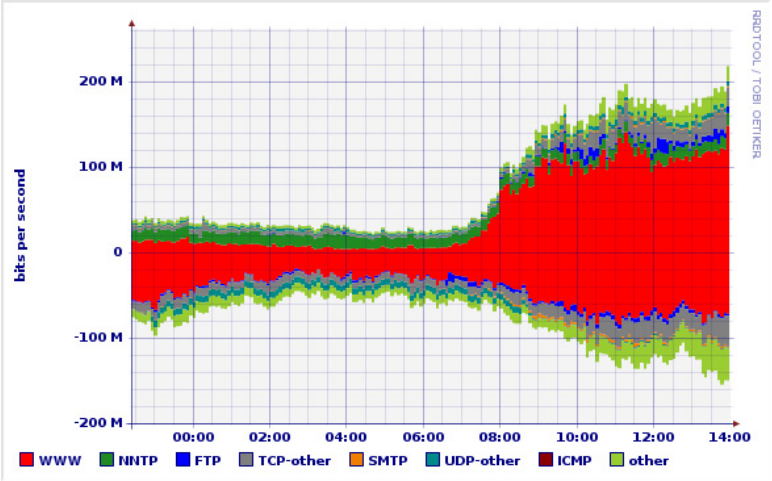

Figure 3 – FlowScan Graph of individual traffic flows inbound and outbound

Network Topology Diagrams

Network Management Systems (NMS) are applications dedicated to the management of networks. They present an abstraction of the network being administered and are typically not concerned with all the details of the network. As such their representation of the network conveys key elements of data visually through the use of network topology diagrams. The data such NMSs display aims to be as pertinent to a broad idea of what a network administrator wants to see. As such a general visualisation of a network as a multi-tiered tree (or star) like structure can be seen below (Figure 1).

This particular representation uses colour codes to signify the 'health' of each particular network device with green signifying no problems, yellow signifying a non-fatal problem and red signifying a fatal problem. In addition shapes are used to display the type of network element being displayed. What is relevant is the representation of the only two elements in the diagram which have other elements going from them: 'Cloud 1' and 'Switch 4226t'. 'Cloud 1' represents the existence of some indeterminate device (or devices) whereas 'Switch 4226t' an SNMP query-able device is able to report what is connected to it.

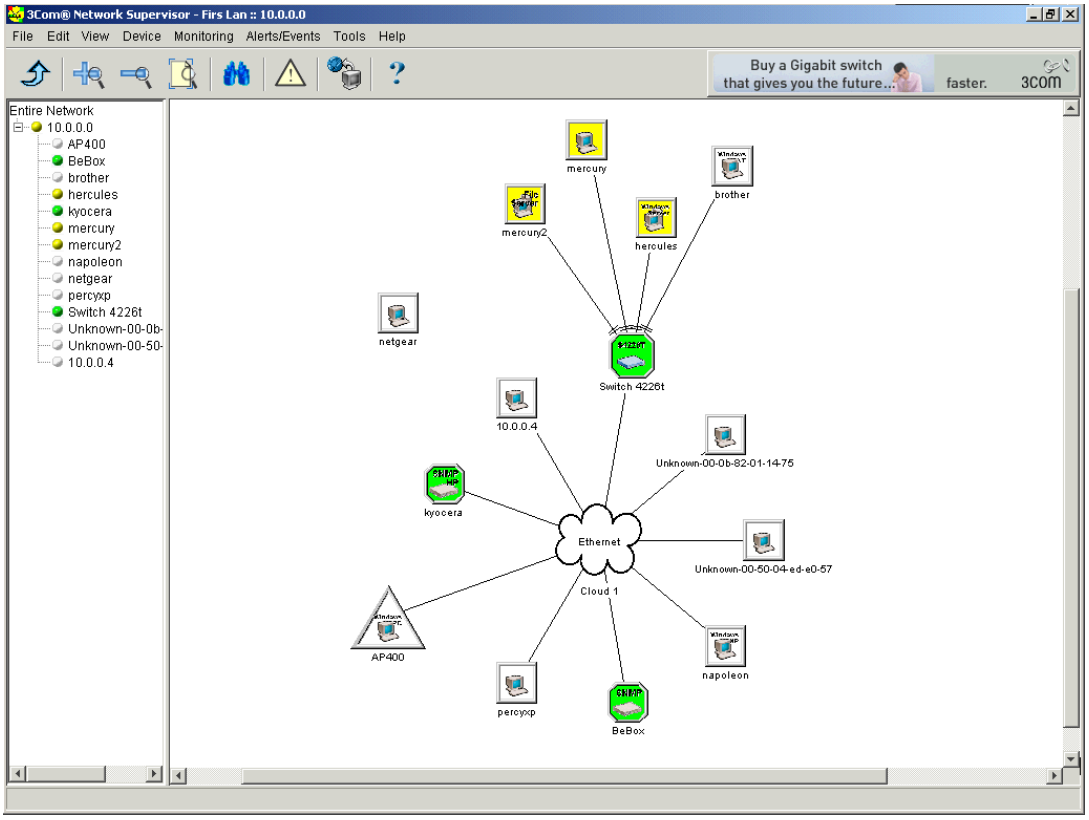

Figure 4 – Network Topology Diagram - 3Com Network Supervisor

2.2 Network Models

To model the network and to store collected information a data 'model' will be required. An exploration of related models follows.

Simple Network Management Protocol (SNMP)

Simple Network Management Protocol[3] provides a means to control and monitor network appliances, allowing configuration management, statistics collection and status monitoring. An industry standard protocol, it uses a tree-like structure (Figure 5) called the 'Management Information Base' in which it stores values pertaining to status, configuration and statistics.

MIB trees are written using Abstract Syntax Notation (ASN) with individual items that hold values making up the leaves of the tree. The tree itself has no limits, it can continue to grow. As such several MIB branches have been standardized and provision for private branches exists so that additional information can be integrated with SNMP tools.

Each MIB value is identified by an object identifier which is a numeric string, but this has a corresponding human-readable value as well. For example the OID 1.3.6.1.2.1.1.3 always returns `sysUptime` from the Agent.

Whilst SNMP is designed for configuring and monitoring devices, the structure of its information base provides a data model that is capable of holding a great deal of information about individual network elements. Using a MIB to provide the basis of the network model will make the model fully interoperable with network devices that are SNMP capable. The application simply populates a MIB-type tree with the corresponding SNMP values that the appliance supports to maintain the network model. Whilst mirroring of the SNMP MIB structure is advantageous in terms of compatibility with SNMP capable devices, with non-SNMP capable devices this advantage is lost. There is an overhead in having to scan a network tree every time a value is required and since the MIB structure contains related information in different sub-trees recalling and storing related data becomes problematic.

Furthermore, the tree like structure of a MIB is designed to occupy a small memory footprint. It does not lend itself well to object-oriented languages such as Java, which is a great disadvantage since an object-oriented approach will be useful in modelling networks in which elements and their data are constantly changing.

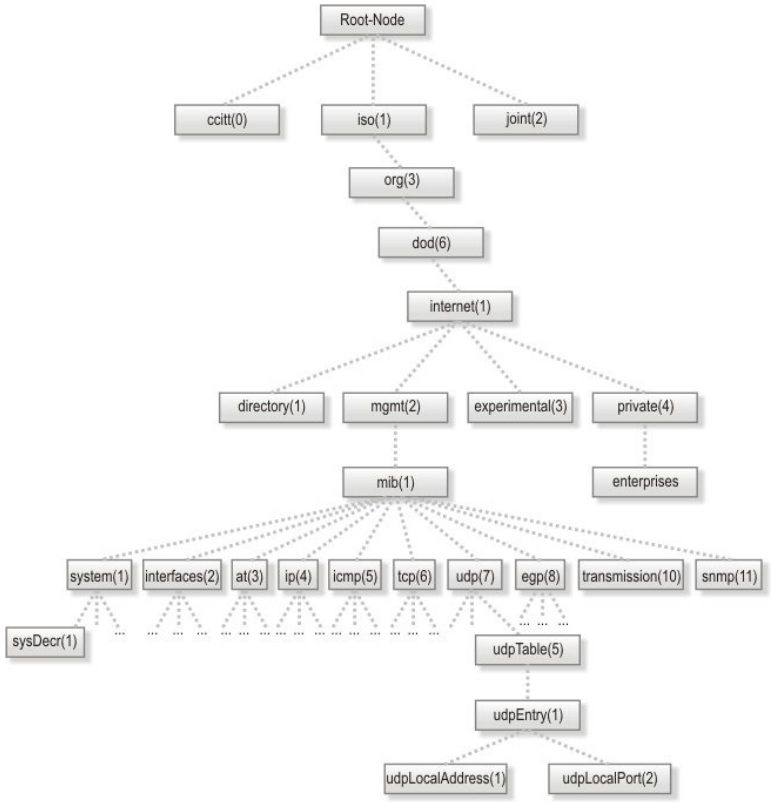

Figure 5 – Management Information Base Structure[1]

Common Information Model (CIM)

The Common Information Model[2] is a standardized, conceptual information model, describing computing, network (and other) entities as well as their environments. It uses object oriented techniques, putting common elements and their methods into groups with associations linking interdependent classes (eg. those between objects that are capable of hosting services and the services that are dependent on them to run).

Within CIM there is a specification and a schema, the specification provides the details for integration of CIM with other management models as well as the syntax of the language definition[2] used to describe CIM elements. The schema describes the individual elements, providing actual model descriptions, methods and rules. CIM defines two models, the Core model which defines the basic structure the management model should follow and the Common model which defines particular areas of management. Such models include Network, Physical and System (to name a few), each group an extension of the Core model.

[1] picture: http://www.xratel.com/snmp_oid.asp
[2] Managed Object Format (MOF)

Figure 6 – Common Information Model – Core Model and Common Models[3]

CIM provides an extremely powerful model with which to design an application which manages networks. It outlines a structure for manageable objects from Computer Systems and the Services that run on them to the Access Points that allow other computers on a network to connect to one another.

CIM also allows for customisation of the model, providing a schema notation with which others can create their own Common Models. By extending the classes CIM already provides one can implement a model that is customised so that it is better suited to individual requirements. The model however is complex, comprising over 230 individual elements in the Network model alone and well over a thousand overall. As such, whilst it offers a powerful and thorough structure in which to store data it will be logistically difficult to maintain.

2.3 Scanning Networks

Nmap

In order to represent networks they need to be scanned. To obtain a list of reachable hosts there are a number of tools available the most popular being Nmap.

Nmap when given a target specification and a query type (such as PING or ICMP), outputs a list of scanned targets. What is useful in Nmap is that it can use a number of different scanning techniques, some of which require privileged access to the machine. It is able to display open ports on a target as well as the MAC address of a target machine. The only problem with Nmap is that it can be very slow when performing port scans and MAC address lookups.

Another problem with Nmap is that if implementing it within the project, it will need to be called using whatever language the project is implemented in. This will require Nmap to be installed on the machine, making the project code less portable since few Windows machines have Nmap installed.

[3] picture: http://www.wbemsolutions.com/tutorials/CIM/cim-schema.html

Nessus

Nessus is a network scanning tool which automates the detection and testing of known security problems. Composed of a client and server, the Nessus server runs on a host machine whilst the client, running on any machine requests the server perform vulnerability scans on a specified range of targets.

The client displays the results returned by the server, which consist of any vulnerabilities found and a recommended course of action if available. The results can be saved in various formats from HTML pages to Comma Separated Values, and MySQL.

Integrating Nessus client functionality with the project code would involve writing a Nessus client from the ground up, or attempting to handle an existing Nessus already installed on the host machine.

Due to the relatively static nature of Nessus's vulnerability results and the long scanning time, the simplest solution would be to populate an SQL database with Nessus results and access that.

DNS Lookups

In order to resolve IP addresses to hostnames so that individual network elements can be given user-friendly labels DNS lookups will need to be performed. Nmap supports this as does nslookup and dig. Unfortunately, not all are installed on every type of machine, and since a cross-platform application is desirable it would be preferable if the implementation language supports DNS lookups itself.

The Java class `InetAddress` possesses this functionality, taking as a parameter an IP address and performing reverse DNS lookups in order to obtain the DNS name.

Simple Network Management Protocol (SNMP)

In addition to the Management Information Base (MIB) SNMP uses, it is first and foremost a protocol designed to allow administrators to monitor and configure devices through a SNMP manager using operations such as Get and Set to query or configure an agent.

The Protocol Data Unit (PDU) specifies to the recipient what operation to perform as well as the object instances involved in the operation. Each SNMP capable network device contains the agent, which is responsible for responding to SNMP operations. To control access to an SNMP device so that no unauthorised modification or monitoring can be performed on it, a community name is assigned it acting as a password into the device.

SNMP offers higher levels of authentication using SNMP v2.5 and SNMP v3, however due to lack of availability of equipment which supports these protocols I will not be implementing them.

To Get or Set a specific value on a network device, such as its name or the amount of data it is transferring the Object Identifier (OID), (a unique string of numbers identifying that value) has to be sent in the Get or Set operation. These OID values are predefined to show a certain type or piece of information. In addition to single values an OID can refer to a table such as the `ifTable` at 1.3.6.1.2.1.2.2.

This table is of particular interest, since it contains information about the interfaces of the device, including the total number of octets received on the interface. Such tables have a predefined suffix to indicate the column in the table with a device assigned suffix to the column suffix indicating the row.

So for example 1.3.6.1.2.1.2.2 is the OID for `ifTable`, with 1.3.6.1.2.1.2.2.1 the index for `ifEntry` (the entries in the table). The `ifEntry` contains a list of values ranging from `ifIndex` (1.3.6.1.2.1.2.2.1.1), `ifDescr` (1.3.6.1.2.1.2.2.1.2) and so on designating the column index. Each row value located at 1.3.6.1.2.1.2.2.1.1.X where X is the row number.

SNMP will be particularly valuable not just for obtaining the description and traffic flows through a device, but for obtaining the topology of a portion of the network. What needs to be noted however is the layer that SNMP capable devices operate at. Layer 2 switches unlike layer 3 switches user layer 2 headers composed of physical addresses, not IP addresses. As a result, if the network is to be scanned by IP, to represent the network topology a translation between MAC and IP will need to be performed.

Syslog

Syslog like SNMP allows a network appliance to communicate information about itself with a network management point. The network appliance generates messages which are sent to a collector or syslog server. These messages are logs from the network appliance and are normally sent in cleartext.

Syslog's key strength is that it is very widely supported, however for a collector to receive syslog messages from an appliance, each appliance needs to be directed to the syslog server.

2.4 *Exploring Ideas*

Topology Visualisation

The network topology diagram below uses three dimensions to display the network topology as well as shape to signify a network element's type and colour to signify an its health.

Here two network routers are represented as pyramids, with individual (child) elements (eg. computers) as spheres coming from them. The black lines between the elements represent an active connection that exists between the child and its corresponding parent whilst the colour of each element represents its health.

Green signifies that the element's health is normal or healthy, whilst yellow signifies a non-critical problem red a critical problem. Utilising conventional colours, (ie. the 'traffic light' colours) the visualisation below takes advantage of learned information that the user already possesses.

What the diagram below shows that is novel is the flow of traffic between the green pyramid and the red pyramid. At a glance, a user can tell what element is connected to what, which elements have problems and which links have activity occurring.

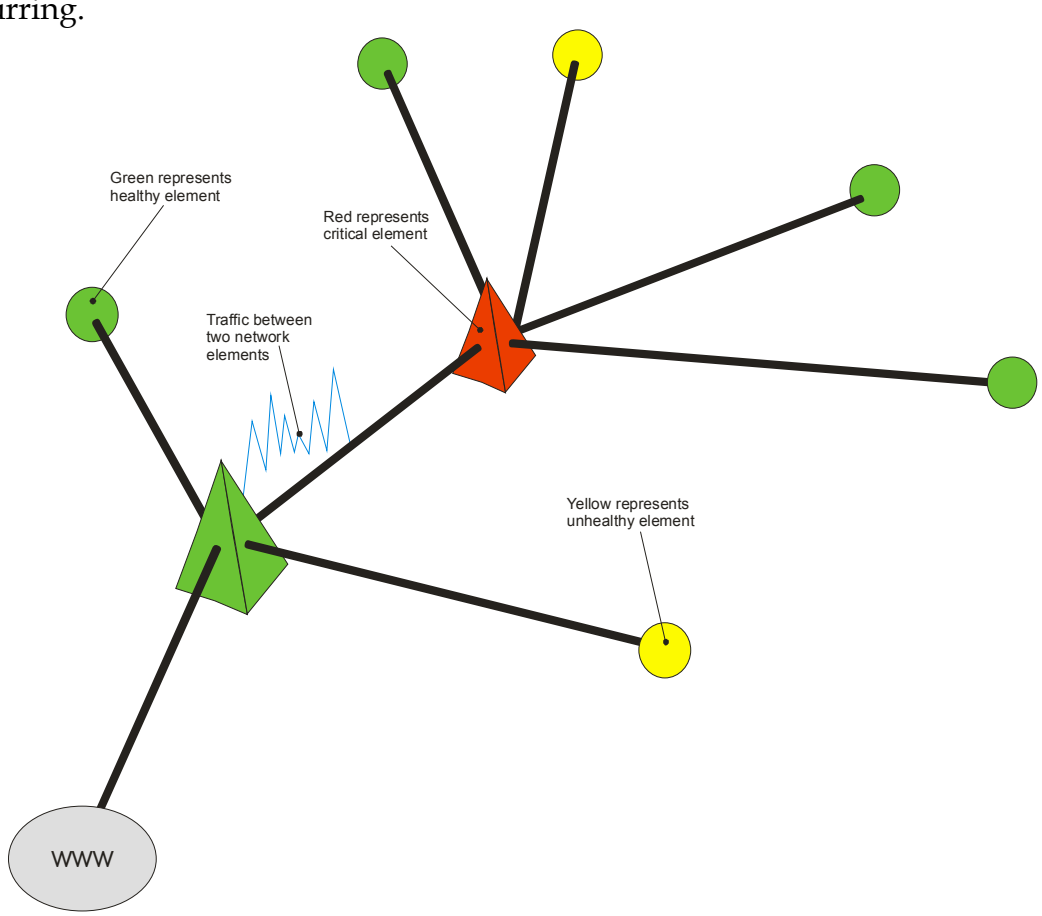

Figure 7 – Three-dimensional network topology diagram with colour conveying status

Traffic Visualisation

In the example below, outlining the decreasing levels of abstraction steps 1 to 3, (or increasing steps 3 to 1) the client-side application is displaying SMTP traffic from the internet to a network router. The traffic into the router is inbound in the positive Y direction (above the line) and outbound in the negative Y direction (below the line). The inbound traffic is marked green because no security problem is deemed to exist within it however the outbound traffic is marked red because of some sort of issue within that.

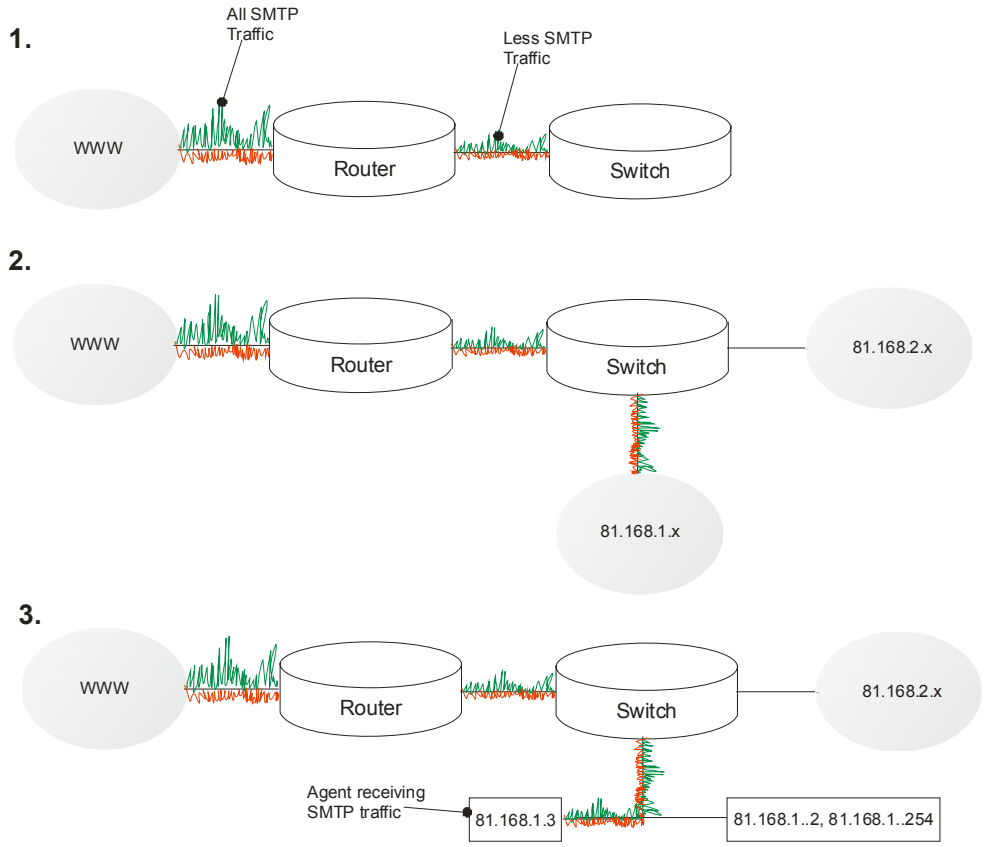

Figure 8 – Increasing levels of detail where user is focusing on where

The representation makes the user aware that the amount of traffic entering the router is more than that leaving the router eg. due to some filtering employed. The user is able to follow the traffic to the switch and then to more detailed representations of the network as that particular traffic is followed in step 2. The user can see at once that subnet 81.168.2.x has no SMTP traffic going to it but that all SMTP traffic is going to subnet 81.168.2.x.

In step 3 the diagram clearly shows that one machine at IP 81.168.1.3 is responsible for all the SMTP traffic across the whole network. The outbound traffic from it flagged red, suggesting that traffic should not be coming from that machine or that it consists of bad email.

3 Design

This chapter details the design and the issues involved as well as the functionality planned and justifications for each component.

3.1 Implementation language

The chosen language is Java due to the inherent benefits it provides:

- It is cross-platform and so satisfies the criteria that the client-side application be operating system independent
- The client-side component can be run as Java Applet, allowing the application to be accessible from any Java supporting web browser.
- Java has a hardware accelerated 3D library (Java3D) which allows integration of three-dimensional elements within the client-side application.

C#/C++ and OpenGL were considered alongside Java, but whilst the former offer excellent speed and response, not having to deal with the overhead of a virtual machine, they would not be runnable through a web browser. I also have more experience with Java than I do with C++ and OpenGL.

3.2 System Structure

The application will be structured into four distinct components, each responsible for a particular role within the overall system:

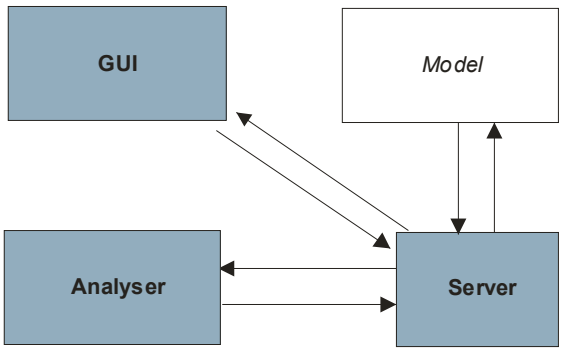

Figure 9 – The four Java packages and their interaction

The server-side component of the system is split into two components; the Server and the Analyser. The Analyser parses the information fed into it and stores it in a database, whilst the Server queries the database and the Analyser with its criteria, populating the model with this information.

The Graphical User Interface (GUI) contains the main method and instantiates the

Server class, requesting the Model via it.

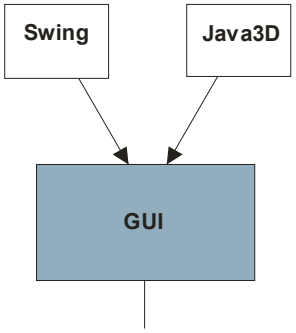

Figure 10 – Graphical user interface composed of Java Swing and Java3D components

The diagram above shows the two components of the graphical interface: the Java Swing GUI and a Java3D GUI. They will be explained in detail later in this chapter.

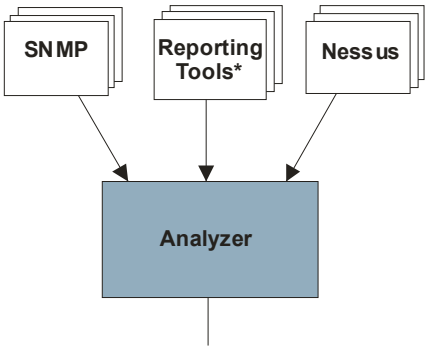

Figure 11 – Input sources for the collector to analyse and tabulate before passing to the Analyser
**(network security reporting tools such as Nessus, CyberCop, etc.)*

The job of the analyzer is to handle multiple sources of network information, these would consist of SNMP values, Syslog messages, and output from network security reporting utilities like Nessus. The analyzer handles the input, parsing it, stores Nessus information in a database and passes the rest to the `Server` class. At this stage I believe it would be best to put relatively static information into the database such as Nessus logs. This is primarily because data which does not change very often does not need to be constantly accessed from each device so the amount of network traffic involved in querying each device is minimised. Another reason for storing data in a database is that some reporting tools such as Nessus take a considerable amount of time to scan a network, so to prevent having to rescan the network every time some data will be stored.

`Server` will sift through the data available from `Analyzer`, taking the relevant parts and populate the network model with it. Once the model is populated the GUI will be notified by the `Server` and update the display as the information changes.

3.3 *Server*

The `Server` class is so called because the original intention was to have the GUI `ClientApp,` run through a web browser and communicate with the backend `Server` via Java sockets. This is why `Server` exists in its own package – additional classes would be added to allow Server to communicate with multiple views via sockets. However, this functionality is not critical to the operation of the program and would have slowed it down somewhat, as well as requiring a significant amount of time to implement. Thus due to the non-critical nature of such a feature the Java sockets functionality was excised and the classes were implemented with their communication occurring directly with one another.

`Server` is the only class to modify the model directly. This was a conscious decision that was made early on; the centralisation of code aiding maintainability of the network application. If the model was modified from a number of classes the maintenance problems involved in debugging and altering the model would have been made more difficult.

To scan the network `Server` uses the class `NetworkScan`. This queries the range of IP addresses the user supplies using the Java class `InetAddress` which provides such functionality as ICMP and ECHO pings as well as reverse DNS lookups.

It was also decided upon that the `Server` class would not be aware of the GUI, being part of the 'Model' component of the Model-View-Controller pattern. In doing this I decoupled `Server` from the implementation of the GUI, aiding reusability of its code, and allowing implementation of multiple views of the same data if so required.

Server utilizes the `Analyzer` package, using the classes `NessusDatabaseManager` to perform operations on the SQL database containing Nessus logs. Whilst these classes could be all integrated into the `Server` package, they are not directly dependent on `Server` for their functionality and so are kept in `Analyzer`.

Since each element is stored in the model using its IP address (and DNS name if it exists) as a key, obtaining information from layer 2 switches regarding which port on the switch is connected to which network IP, is difficult. Layer 2 switches route packets using layer 2 headers so interrogating each port via SNMP to see what network element it is connected to simply returns the element's physical address (MAC address). The IP address of a device cannot be looked up using its MAC address with `InetAddress`, so the interrogation of ARP tables on the switch and any other SNMP capable devices will be performed to populate the model with as many MAC to IP address associations as possible. What should be noted however, is that if an element has no ARP entry on an SNMP queryable device, its connectivity information (ie. what it is connected to) might be wrong.

Because the network switch I was querying is a layer 2 switch, its header information consists only of physical addresses (MAC addresses). Since element's are identified by their IP addresses and DNS names, the only way of obtaining the

corresponding IP address for the MAC address each port on the switch is connected to is by querying the ARP table of the router. If the router does not contain an element the switch is connected to, the element does not appear as connected to the switch.

3.4 Analyser

The information base (MIB) Simple Network Management Protocol (SNMP) uses was explored initially as a basis for the network model. Although it was discounted in favour of CIM, SNMP is invaluable in other respects.

To write libraries which obtain data via SNMP using Java is a project in itself so external libraries were used. To do this I went straight to the only open-source SNMP Java stacks I could find, Westhawk and SNMP4J.

Westhawk – Westhawk is an SNMP stack which offers support for sending `Get`, `GetNext` and `Set` requests. In addition it is able to deal with traps. A disadvantage of the Westhawk SNMP stack is that it does not have any MIB browser functionality. Thus, to translate an OID to a human-readable representation will require an external MIB browser. Westhawk implements its SNMP operations using threads. As a result, whilst this makes coding of events to listen for the operations to complete a little more tricky, it does mean that the GUI will not be blocked whilst the SNMP operation completes. Westhawk is also well documented and the older of the two SNMP stacks evaluated.

SNMP4J – SNMP4J offers the same functionality as Westhawk, but is a younger stack and less developed. It is also not as well documented as Westhawk.

Since both stacks offer much the same functionality, Westhawk's better documentation and maturity led to it being chosen.

In addition to the Westhawk SNMP stack, I used a Management Information Base (MIB) parser called Mibble[4]. A MIB uses a tree-structured naming convention that defines what a particular OID number represents. For example the aforementioned OID 1.3.6.1.2.1.1.1 is stored in the agent alongside its value (eg. 'Telco IOS 1.11 Device'). The OID number does not tell us what that value signifies, but using a MIB file the number can be looked up to obtain the human readable name `sysDescr`.

This is to be implemented in the class `SnmpHandler` which will handle all SNMP `Get` requests and load the necessary MIB files.

To utilise information from Nessus, an external Nessus client and server are used to scan the network before hand and populate a MySQL server with the records. These records are accessed by the class in the `Analyzer` package:

[4] Mibble - http://www.mibble.org/

`NessusDatabaseManager`. This class is accessed by `Server` which is able to populate the model with this information using the `Analyzer` package.

3.5 Network Model

To represent the network in computer memory, a model will have to be used. The model is one of the most important components and its choice is of critical importance. The model should ideally provide:

- **Object-oriented techniques –** The model would need to provide a hierarchical structure, allowing the sub-classing of root-classes which provide basic functionality common to all components. This would be required to deal with the variety of network components being modelled - many having similar fields and methods but different capabilities. In addition loosely coupled signals and checked get and set methods will ensure the model is adaptable and well designed.

- **Modularity and extensibility –** Being able to extend from common classes to include customised behaviours as well as maintaining consistency at higher levels of the model makes the information more manageable.

- **Standards based format –** The use of a standardized structure to define the model information is important. The model is likely to be better documented as well as being easier to integrate with other applications and devices which support the same model. Models which are standardized tend to have compatibility in minds so that they map with other information models (such as SNMP).

Of the two models explored, Simple Network Management Protocol's and the Common Information Model, the latter was by far the best. Using object-oriented techniques it lends itself best to the chose language, Java, and also encompasses not only individual network elements (as SNMP does) but the network in its entirety from network links, to the physical connectors that terminate them.

CIM uses extensibility to define different classes of elements, the root class `ManagedElement` extended by `ManagedSystemElement` and this in turn extended further. `ManagedSystemElement` represents systems and system components these include computer systems, services and network interfaces as well as many others. From `ManagedElement` are a number of associations like Dependency and Component, which are used for defining additional components of a system.

Sub-classes of `ManagedElement` have associations more relevant to particular devices (such as the association `HostedAccessPoint` which exists between `ServiceAccessPoint` and the `System` which hosts it) making the generic associations in `ManagedElement` unnecessary in my implementation.

In order to represent network topologies using CIM an awareness of both the logical representation of the network structure and its physical representation are required. CIM uses the Physical model to map physical attributes onto logical models (Network in my case).

3.6 Swing Graphical User Interface

Whilst Java3D was settled upon for visualising of network topology diagrams and network traffic, it was found to be merely adequate in representation of text and cumbersome at best with implementing buttons in three-dimensions. As a result, Swing[4], Java's general GUI toolkit was decided upon. With Swing, one can design interfaces with tree components, tables, tabbed dialogs, tooltips, and several other GUI features. Because of its strong functionality in dealing with two-dimensional graphics and GUI components Swing is used to handle text field entry, Windows, Panes, Tables and Buttons.

The aim of the project is to display information in the network topology diagram wherever possible. However this is not always feasible, for example the display of Nessus logs requires the use of a table. Because of the inferior quality of 3D text in Java3D, Swing is required to display high-quality, readable text.

The Swing GUI class `MainWindow`, will contain the main method of the program. This is because Swing is capable of holding a heavyweight component (ie. Java3D) within it whilst Java3D cannot hold a Swing component. In addition, since the program is client driven, having `MainWindow` instantiate the `Server` class is logical since `MainWindow` initiates the scan based upon the user's input.

3.7 Java3D Graphical User Interface

Researching the Client-Side Applet

Network topology diagrams typically consist of a multi-tiered tree structures presented in two-dimensions. This conveys the connections between nodes as lines and represents further information through the use of icons to represent different types of nodes. However, spatially, in the physical sense, a network is in three-dimensions, and the capacity to convey information in three-dimensions is certainly greater than its two-dimensional counterpart. An important point to bear in mind is that whilst three-dimensional representations of small networks are capable of easily disseminating information to t he viewer, larger networks can appear cluttered and unwieldy.

To avoid this, the system should present the user with only as much information as they wish to see, ensuring the interface remains as uncluttered as possible. Alternatively, the network can be presented using a two-dimensional network

topology, with the third dimension being used to display information that would be difficult, or impossible to convey without it.

The language and libraries I need to use to construct such an interface ought to satisfy the following criteria:

- High performance – To represent ever larger networks, and their individual network elements a relatively fast library needs to be used. If the interface proves too slow, it detracts from the user experience as well as limiting the visualization tool in how much information it can present graphically.

- Integration with widgets – The 3D elements of the system should be capable of integrating with the standard GUI widgets that the windowing environment provides.

- Object-oriented – To better integrate the GUI with the existing choice of Java for the information model and backend components an object oriented API would best suite the chosen design.

I used these criteria to evaluate the following GUI libraries:

Java Swing – Swing can be run as applets within a web browser and is cross-platform requiring only the JRE to be installed on a client system. It would provide a flexible and customizable user interface which is particularly important when creating a series of dynamic views of a network. It is scalable and has high performance, coping well with numbers of network elements represented on the screen at once. It also provides GUI widgets along side its drawing library making it capable of the windowing and three-dimensional aspects of the API. However it is inherently two dimensional and so situations involving three dimensional space would be difficult to represent.

Anfy3D – A 3D rendered written in Java, Anfy3D runs well on ordinary hardware and is scalable allowing it to cope with many elements on screen at once. In addition it relies on just the JRE to be installed on a client's system making it more compatible with existing systems. However documentation is less than the other APIs. It is also proprietary in nature and although powerful has few libraries.

Java3D – an official Sun 3D standard, Java3D came under some criticism in its early stages but has recently become a more viable 3D API. It is cross platform, and powerful, with a large number of libraries and documentation. Java3D can also be combined with Java Swing to provide a 3D canvas in addition to the windowing widgets that Swing provides. It is heavyweight (Swing being lightweight) so any Swing components are drawn under Java3D components. Because Java3D is a high level API it can be slow when rendering busy scenes. In addition it also needs a set of libraries to be installed in addition to the standard

JRE, however these were easy to install on Linux and very easy to install on Windows.

I have decided to choose Java3D primarily because it is Java and thus easily combined with the Java libraries. It is also one of the best supported 3D APIs, with an active open source developer community around it. Its support of hardware acceleration, a near mandatory requirement for 3D libraries, as well as its strong set of libraries and documentation makes it by far the best candidate to suit the chosen criteria.

Background – GUI

Java3D implements a 3D environment as a Universe composed of two main branches, the Content Branch and the View Branch. The view branch comprises the user's perspective of the scene and contains the `Canvas3D` which renders the view, the `ViewPlatform` which controls the position of the viewer and the Screen information as well as the capability to handle multiple screens. Apart from the American spelling of every object in the graph, the Universe tree is relatively simple to get to grips with, with the scene elements extending the class Nodes. The main group is `BranchGroup` to which TransformGroups, Behaviors and Shapes are added. `TransformGroup` holds a transform which applies to any children below it. For example a rotation behavior (to make a gear rotate) is applied to the `TransformGroup` containing the gear shape. Anything above that `TransformGroup` (for example the gear housing) does not rotate with it. Behaviors also apply to all subgroups below the Group and handle an event such as a rotation event (ie. a spinning gear) or a mouse or keyboard event. Behaviors do not apply to a Node unless the appropriate permission is set: eg. for a gear to rotate gear the `ENABLE_TRANSFORM_WRITE` bit has to be set, for an object to be pickable `setPickable()` has to be true.

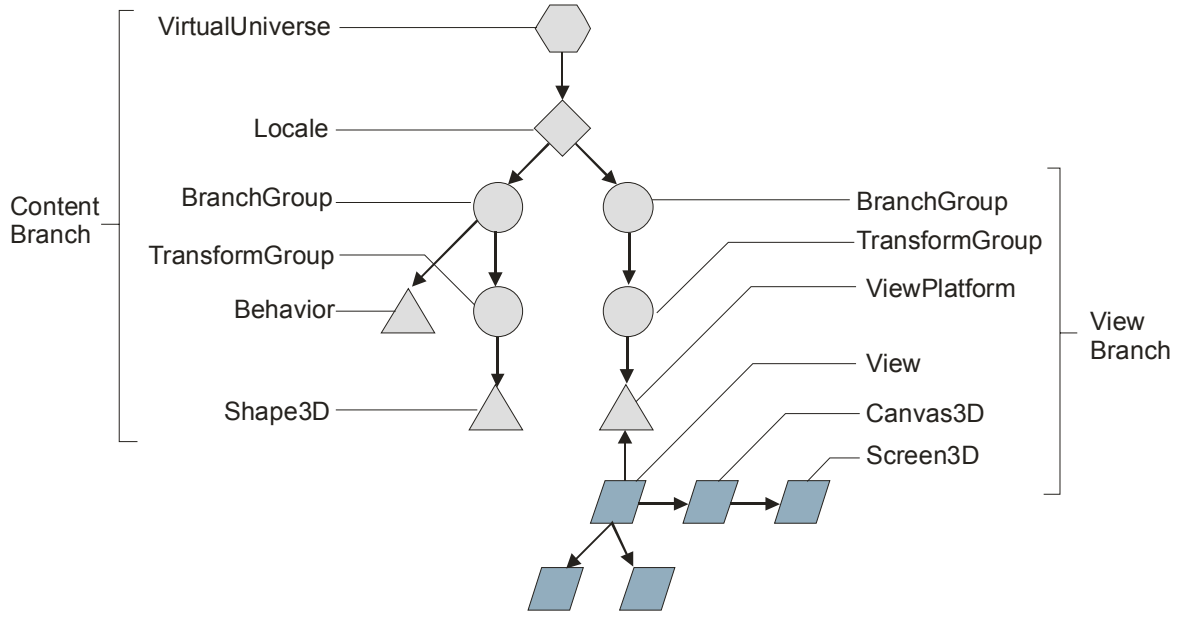

Figure 12 – Diagram of View and Scene Branches

These permissions are set prior to run-time due to Java3D performing internal compilation of the scene; any unset capabilities are removed from the objects to conserve system resources.

Experimentation

In order to show the name of each network element a token, or label of some sort had to be displayed. By far the most legible labels were produced using rasters; comprised of pre-drawn images rasters are images applied to a polygon so that the image appears floating in three-dimensional space. The first problem with this was that a label would be required for each network element, and as the number of elements increased, so more labels would have to be drawn-by hand and saved as images which would make the implementation very inflexible when new network elements are added. The second problem with rasters, was that they obscured whatever lay behind them, and when viewing the network view this meant that anything behind the raster would be hidden. What worked better was using Text3D shapes, which could represent any given String before or after run-time.

4 Implementation

This chapter details how the system is implemented, describing technical decisions and any hurdles that had to be overcome in the process.

4.1 Server

The `Server` class controls the main components which provide access to Nessus data and SNMP queries. In addition it constructs the model, scanning the network for reachable IP addresses and assigning them corresponding elements in the model.

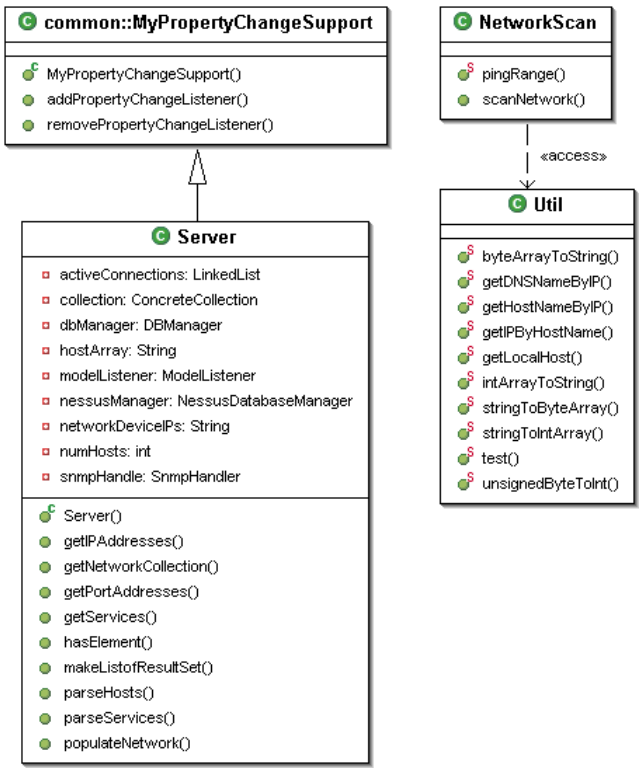

Figure 13 – UML Diagram of Server package

The method `populateNetwork()` in `Server` is invoked by `MainWindow`, the central Swing GUI class. Given the parameters start IP and end IP, `populateNetwork` calls the method `pingRange()` in the class `NetworkScan`, passing it the same arguments. The method `pingRange()` returns an array of `InetAddress`. `InetAddress` is a class which represents an IP address and allows pings and reverse name lookups to be performed on the IP – the array contains only reachable IP addresses so that `populateNetwork()` constructs a model of active network systems. The elements in the `InetAddress` array are used to construct `IPProtocolEndpoint` elements which are given the same IP as their corresponding `InetAddress`. These endpoints are referenced to by a `ComputerSystem` element which is given the name returned by

the reverse DNS lookup performed by `InetAddress`. The element's IP acts as a key to identify it in conjunction with its name.

Using the array of reachable IP addresses, `Server` instantiates the `NessusDatabaseManager` class with the corresponding hostname from `InetAddress`. Through `DBManager` which handles the database connection, `NessusDatabaseManager` accesses an SQL database to obtain a `ResultSet` of Nessus records. I implemented `DBManager` to provide access to PostgreSQL servers originally, but the programme I used to query my Nessus server and upload the results into a database would not work with anything but MySQL. As a result `DBManager` is capable of handling both SQL servers. `NessusDatabaseManager` parses the `ResultSet` to obtain any Nesuss records stored. For any results it finds which match the given host name it adds them to a `LinkedList` of `NessusResult` which it returns. The Server class adds this `LinkedList` to the `ComputerSystem`.

In addition to `NessusResult`, `SNMPService` is added to those `ComputerSystems` which are SNMP capable. `SNMPService` itself queries automatically whether a device it is assigned to supports SNMP. If it does then its method `isReachable()` returns true.

In the methods `getPortAddresses()` and `getIPAddresses()` Server queries any switches and routers which are SNMP capable. The first method populates switches with the `ServiceAccessPoint`; `SwitchPorts`, this defines the ports on a switch. The second method obtains the list of IP addresses and corresponding MAC addresses in the ARP tables of SNMP capable devices so that the two can be tallied. Once the two are matched, a `LANEndPoint` holding the value of the MAC address is assigned to the `IPProtocolEndpoint`. This allows the model to be able to show which element the switch's ports are connected to.

The Server class implements property change reporting to Listeners and also contains an inner-class; `ModelListener` which listens for a response from `SNMPService` as to when its SNMP query is complete. Server needs to be able to fire a `propertyChange()` notification so that its `Listener`, `MainWindow` can tell the GUI when to update itself to reflect changes to the model. Server thus acts as a Listener and constructor of the model.

4.2 Network Model

To represent a network I used the Common Information Model (CIM), and in particular the two Common models within the specification: System and Network. The chosen implementation language, Java is a logical companion to CIM – the two sharing object-oriented techniques, (such as inheritance and encapsulation). CIM schema classes map onto their corresponding Java classes easily, their fields and associations intact. To implement the network, individual network systems (such as switches and routers) had to be modelled first.

Network systems are modelled as an instance of `ComputerSystem`, being themselves a computer system with the additional services and service access points that make them network devices.

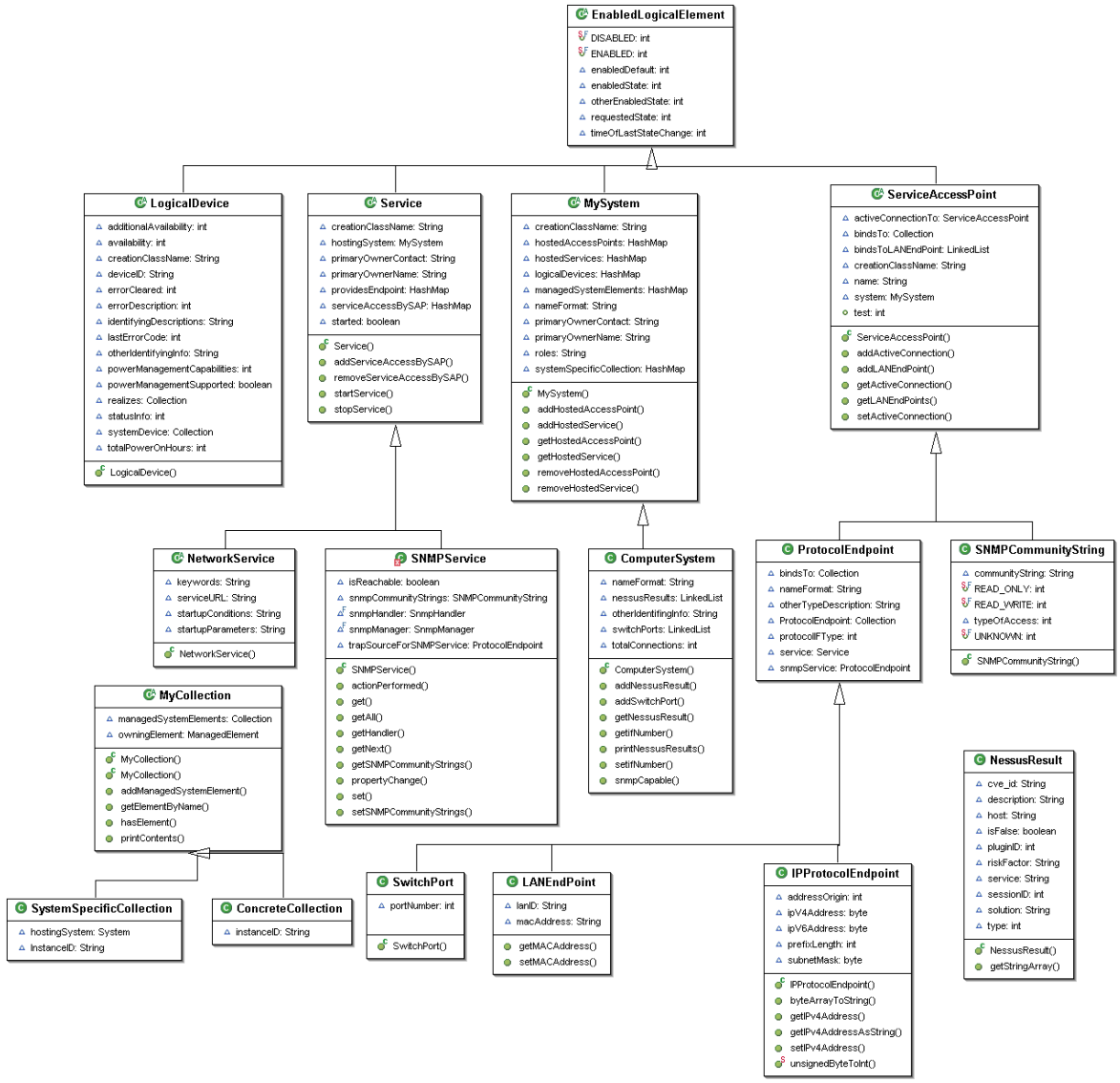

Figure 14 – UML Diagram of network model package

Network elements are accessed via `ProtocolEndpoint`, a subclass of `ServiceAccessPoint`. For example to access a network device over an IP based network the class `IPProtocolEndpoint` would be used. `IPProtocolEndpoint` extends `ProtocolEndpoint` and describing the IP address of the device as well as its subnet mask. It inherits the fields of `ProtocolEndpoint`, which provides details on bandwidth, keepalive timers, interface type etc. For example the field `ProtocolIFType` is a descriptor containing the interface type with its values predefined (ADSL, SDSL, CATV Upstream, etc.) Through this the network element's Services can be accessed.

Two ProtocolEndPoints are associated via the many-to-many relationship, `BindsTo`, allowing for single connection links as well as multiple connection links, eg. using bonding. Two active ServiceAccessPoints are represented by the association `ActiveConnection`. This association represents an unmanaged

connection, with another hierarchy of association representing a managed one (and all that it entails). For simplicity's sake I have implemented the `ActiveConnection` association as unmanaged by default, since this capability is not needed to display traffic flows. In addition instead of representing an `ActiveConnection` as a link between two ServiceAccessPoints I have implemented it as a reference called `activeConnection` within `ServiceAccessPoint`. Thus any `ServiceAccessPoint` can have an active connection with another.

The most complex class in network model is `SNMPService` incorporating within it the capability to query an SNMP device. I wanted to keep the model as simple as possible, with no capability in any class to query the network. This was so that the model would be completely implementation neutral, being simply a repository for values that together represent a network. I did this for all the classes in the model except `SNMPService`. This is because implementing a complete representation would have required every single SNMP value stored in the device to be stored in the class. This would have required a tree of more than 1000 values on devices such as switches, and would require `SNMPService` to be updated constantly with new values so that it maintains an accurate representation of the network. Unnecessary network traffic would be generated and the network devices being queried would be slowed down (some of which in my testing proved to have very slow SNMP responses). As a result `SNMPService` was written to have the capability to perform queries for SNMP values so that they can be looked up as needed. The disadvantage of this is that any class that wishes to obtain a new SNMP value must implement a Listener so that it can receive a notification when the query does or does not return.

`SNMPService` itself implements `PropertyChangeListener` so that when the SNMP query is ready it is notified. In addition `SNMPService` is capable of firing property change notifications to potential listeners so that they can be notified when their query is ready. This capability was added to the top most model class `ManagedElement` (the only model class not extending a super-class), by having it extend `MyPropertyChangeSupport`. As a result every model element is capable of notifying Listeners that a change has been made to them.

The Physical model can be used to add additional data for each logical element so that it relates to the physical components from which it is comprised. The `PhysicalElement` class contains this information and is associated with its corresponding logical class via the Location class. With information about a network element, such as the rack space occupied, its physical location and individual components, the model allows for very detailed representation of a network domain. The scope for visualising a network from the physical to the logical level, whilst too ambitious to implement here, is not unfeasible.

4.3 Simple Network Management Protocol

SNMP functionality is contained in the `SnmpHandler` class which is given the parameters host name and community name for the device. Implementing a

Listener class it listens for SNMP responses to fired SNMP Get queries, which query the SNMP Agent across the network. Once it receives a response it fires a `propertyChange()` so that any Listeners can be notified. The only listener of `SnmpHandler` is `SNMPService`.

To send a Get query a `SnmpContext` has to be created. The context is built from the host name to which the PDU has to be sent as well as the community name that is used. Contexts control SNMP communication between the manager and agent, and can be reused to perform more than one operation. As a result they must be destroyed when they are no longer needed so that Java can perform garbage collection on them.

In order to translate OID numbers into MIB values `SnmpHandler` loads a mib file, passing it to an instance of `Mib` (from the Mibble package). The method `getSymbolByOid()` in `SnmpHandler` is given the OID as a parameter and after loading the default MIB file into `Mib`, queries are performed to look up the human readable value of the OID.

4.4 Swing Graphical User Interface

The main method of the application resides in the class `MainWindow`. This calls the method `createAndShowGUI()` which creates a new instance of `MainWindow` and adds the `JSplitPane` that `MainWindow` creates on instantiation. The split pane consists of two JPanels, one containing a `JTabbedPane` which holds Swing GUI elements the other containing the Java3D view. To combine Java3D elements with Swing the Canvas3D in `ClientApp` (the Java3D class) has to be added to a JPanel. Since Java3D is a heavyweight element and Swing lightweight, any `Swing` elements that are drawn appear under Java3D elements.
The constructor `MainWindow()` also creates an instance of the Server class, adding an inner-class `ServerListener` (which implements `PropertyChangeListener`) to Server's list of listeners. Upon `Server` calling `firePropertyChange()`, which notifies `MainWindow` that a change has occurred, `ServerListener` calls the method in `ClientApp`; `updateCollectionGroup()` – this updates the three-dimensional view passing it the latest model (`ConcreteCollection`) as a parameter.
The `JSplitPane` which holds the Swing elements is the first `JPanel` the user will see. It is composed of two text fields and a button and is constructed by the method `createStartPanel()`. The user can enter an IP address range here, providing a simple interface via which the network can be scanned. To do this, `MainWindow` adds a new `ActionListener` to the button which invokes the network scan. This listener is implemented as an inner-class since its scope is confined to the 'start scan' button and it calls the method in Server; `populateNetwork()` with the two text fields 'start IP' and 'end IP' as parameters.
Once the scan is completed by Server it calls `firePropertyChange()` which notifies `ServerListener` to update the GUI.

4.5 *Java3D Graphical User Interface*

Implementation

The class ClientApp is the main class for the Java3D component of the project. Receiving from MainWindow the network model, it parses the model elements to produce a three-dimensional representation.

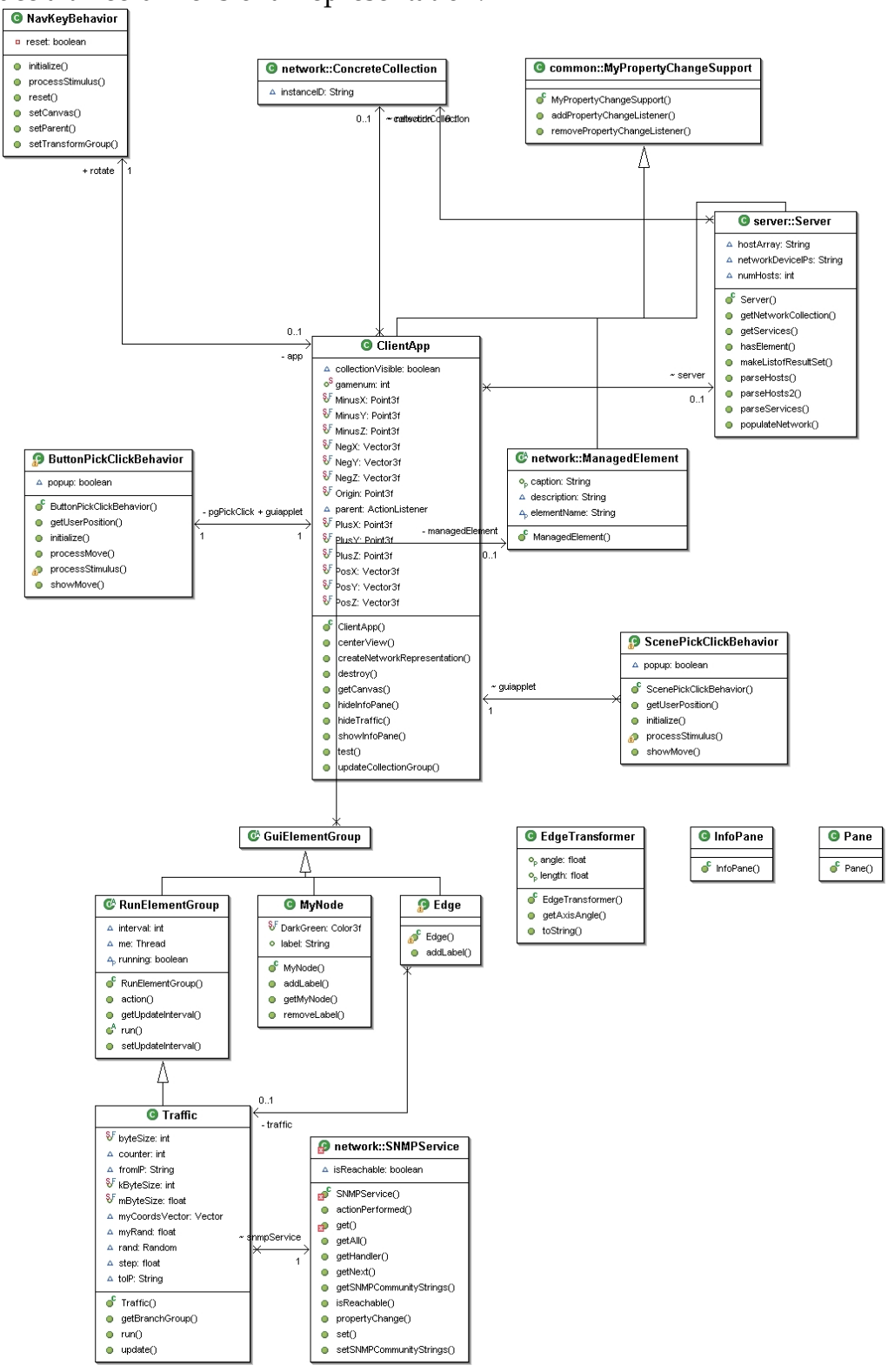

Figure 15 – UML Diagram of Java3D Package

The initial construction of the Java3D scene is achieved by utilising the Java3D helper class `SimpleUniverse` to setup a minimal user environment. Containing a `ViewingPlatform`, `Viewer`, and `Locale` it offers single view functionality. For the case of additional displays and multiple views it can be customised, but for my purposes it is suitable. To define the user's initial view the `ViewingPlatform` is modified to alter the view of the scene from its default coordinates (0,0,0). To do this the `TransformGroup` in `ViewingPlatform` is obtained using `getViewPlatformTransform()` and the `Transform3D` (defining a translation in 3D space) is applied. I found that a bird's eye view of the scene or a isometric view in the direction of the origin worked well.

In order to add graphical elements to create a 'dashboard' for any user notifications and information panes the `PlatformGeometry` class (an extension of `BranchGroup`) is added to the `ViewingPlatform`. The two capability bits `ALLOW_CHILDREN_EXTEND` and `ALLOW_CHILDREN_WRITE` need to be set to enable any additions to `PlatformGeometry` to be made after run-time.

The two `Behavior` classes, `ScenePickBehavior` and `DashboardPickBehavior` are added to their respective BranchGroups. `ScenePickBehavior` is added to the content `BranchGroup` and `DashboardPickBehavior` is added to the `PlatformGeometry` group. These two classes handle picking in their respective parents, I go into more detail about them later.

The content `BranchGroup` is returned by the method `createSceneGraph()` in which the content data (such as shapes, lights, sounds etc.) all reside. The initial elements this method adds to the content branch provide the initial environment so that objects can be viewed, and the background colour set to white. The capability bits `ALLOW_CHILDREN_EXTEND` and `ALLOW_CHILDREN_WRITE` are set to allow any additions to the content `BranchGroup` after run-time.

The background is set to white and its bounds, a closed volume defining where the implementation may disregard the processing of any elements beyond the spatial scope of a target object, is also defined. Most importantly in the method three lights are added, which without there would be no illumination to see any objects by. Two of the lights directional, allowing for highlights (diffuse and specular) on the surfaces of objects, and one light is ambient, giving a uniform light level throughout the scene. Again the bounds are set, defining the scope of the light objects.

To create the network representation the method `updateCollection()` takes the parameter `ConcreteCollection`. `ConcreteCollection` is a collection of `ManagedSystemElement` (from the CIM schema) and holds the model. The method checks if the network elements have already been added to the content `BranchGroup` by checking a boolean value. If this check is not made there would be overdraw issues with multiple elements occupying the same space and issues with elements that have been removed from the model still present in the view. If the Boolean value is false ie. the network representation has not been added to the

scene, then the method `createNetworkRepresentation()` is called. If the Boolean value is true, then the `BranchGroup` which contains the network representation is removed from its parent, the content `BranchGroup`, and all its children are removed. The network reprentation is then constructed afresh by calling `createNetworkRepresentation()`.

Network elements are assigned a corresponding shape, ie. leaves are represented as spheres, parent nodes as pyramids, and network connections between elements represented as cylinders. Traffic line and area graphs are represented by lines and triangles respectively.

The allocation of points in three-dimensional space to children versus parents required a well defined visualisation concept for the network to be decided upon first. The chosen concept, with links between nodes represents a star-topology network diagram. To allocate coordinates to parent nodes a circumference around the node calculates the maximum and minimum x and y coordinates any leaf can take. The radius of the circumference defining the distance between the parent node and any leaves. Any children of the node which are not leaves (ie. have children coming off them) are given minimum and maximum coordinates of 2x and 2y, so that their children do not overlap the children of the parent node.

Individual nodes are implemented as the class `MyNode` with their corresponding shape according to what type of element they model and a label representing the element's name or IP. For the program to identify which element an individual Shape represents the field `userData()` is set as the element's name. The element label has to be legible, which given text made of polygons is not always the case. To achieve this the labels were made larger, a simple black font used to ensure their legibility - however, the problem with words in three-dimensions is that when viewed from any perspective besides head-on they cannot be read. To overcome this, a new class had to be written, `Label`, containing the label as a `Text3D` shape, held within an `OrientedShape3D` object. The `OrientedShape3D` object holding the `Text3D` shape ensured that the label faces the user no matter which way they view the network.

Links between nodes are implemented using the class `Edge`, where cylinders are drawn between points in space. The translation of the edges into three-dimensional space was less trivial than the nodes, since the edges are not simple points, but have a start point and an end point, requiring each edge to be scaled between the nodes accordingly.

To do this, a class called `EdgeTransformer` had to be written. Taking a start point and end point, the class finds the mid-point between the two, and calculates the length the 3D shape must span. This required trivial Pythagorean maths. What proved tricky was calculating the matrice holding the rotation of the cylinder, so that it would line up between start and end point. To do this, the cross-product between two vectors had to be computed, with the angle in radians applied to the shape to rotate it correctly.

To show the flow of traffic between two network elements in Java3D I used the style of a line and area graph in the class `Traffic`. The graph is drawn between network elements with the *x*-axis along the length of the cylinder shape that represents a connection between two network elements. The class `EdgeTransformer` is used to perform the transform on the `Traffic TransformGroup`. Like `Edge`, `Traffic` is drawn between a starting point and end point. Unlike `Edge` however, `Traffic` is not finite. It can be drawn into infinity so long as there exists a link between two elements. To deal with this `Traffic` is limited to the length of the cylinder it follows, the start of the graph being culled as the end of the graph continues to show new traffic data. Traffic extends Thread and wakes regularly to poll its source for data. The source of the data that Traffic queries comes from a reference to a SNMPService object which SNMP capable ManagedSystemElements contain.

Input Device Handling

It is important to have an easy to use user interface so that the user can navigate the "world" and manipulate objects easily, for example moving the camera angle but not moving it completely outside the scene.

For this view Behaviour's were set up to take input from the cursor keys and mouse, together providing the user a simple interface that does not require a tremendous amount of experience to use.

Navigation

Input device handling in Java3D requires a Behavior class which is able to handle the input device and based upon the data from the device update the display to reflect the user input.

On user input the keyboard or mouse creates an event which has to be handled. Using the data from the event a corresponding operation can be performed, such as a transform to the viewpoint. This makes the behaviour node useful when limiting the user's viewpoint to a certain area of 3D space.

Basic functionality for user navigation of the scene via mouse is achieved using the Java class OrbitBehavior. Using rotation, zoom and translation actions this class allows the user to manipulate the view point to any position within the 3D space. Transforms are made within the ViewGraph to the ViewPlatform and not to the scene containing the individual 3D elements.
To prevent the user zooming so far into the scene that their viewpoint is lost within a 3D object the minimum radius of OrbitBehavior is set. Although the class provides freedom of movement within the 3D space via mouse, navigation using the mouse alone is a little cumbersome.

To counter this an additional Behavior class is implemented, NavKeyBehavior. Providing the user the ability to perform some navigation functions using the keyboard NavKeyBehavior allows the user to rotate and zoom into the scene

using the as well as to reset their viewpoint.

Together the two navigation classes used provide sufficient control to allow a user to change their view to any point they wish.

Selecting Elements

Handling user selection in a three-dimensional coordinate system is required to allow manipulation of individual objects and the display of context sensitive menus as well as information from individual elements.

To implement picking the `Behavior` class is extended. Initially I opted to have one class, `PickBehavior`, handling pick events on both the content branch and the view branch. The advantage of this was having one class to handle pick events, with tighter integration between methods in the same class, and easier pick handling between Content and View branches. However, to decouple the classes and simplify the code I chose to use two classes instead, `ScenePickBehavior` and `DashboardPickBehavior`, one for the content branch and the other for the view branch.

`ScenePickBehavior` is applied to the content branch, handling any objects below it (such as the 3D elements on screen). Because Java3D compiles itself at run time, permissions on individual nodes that I want to be pickable need to be set before hand, since these are disabled by default. Setting the pick state is done through the use of the `setPickable()` method of the `Node` class. This allows any scene graph node to be pickable. The field `ENABLE_PICK_REPORTING` is used to allow a group to be selected when an object within it is also selected. Setting these on individual `Node` groups allows the Sphere within the `Node` group to be picked and the parent group to be returned.

The way in which the pick method selects objects is by projecting a vector from the pick point and calculating its intersection with an object. The object selected would then be returned. The problem with this however is that when dealing with the complex geometric structure of a network, with many different shapes in it, the pick intersection will occasionally select the wrong object. Fortunately, once the correct pick state was set for objects I wanted pickable or non-pickable, this problem became a rare occurrence.

The picking method was non-trivial to construct, requiring handling of mouse events, which would be grabbed from the input stream, and then handled according to their ID. For example a mouse click would have a different ID to a mouse move. Furthermore, a mouse click would be different to a mouse press, or a mouse dragged, or a mouse released, requiring the final implementation of the pick class to be tested with various combinations until picking felt responsive.

Once a shape is picked the field `userData` is checked to obtain the name of the network element the shape represents and context sensitive elements can be added to the Java3D scene.

5 Evaluation

This chapter evaluates the network visualisation tool and the strengths and weaknesses of its approach

The central objective of the project was to produce a network information tool that could scan a network and produce as accurate a three-dimensional representation as possible. In addition the tool had to take advantage of the third-dimension, aiming to integrate data in the topology diagram whenever possible.

These objectives have been largely met, resulting in a capable, easy to use tool which does present network topology in a novel way. The initial scope was to visualise network information at different network layers, from the Application layer down to the Physical layer. Whilst the network model used would support such data, obtaining this information proved extremely difficult. The SNMP switches used were layer 2 switches and thus only able to display layer 2 header information (MAC addresses). As a result physical addresses had to be translated to IP addresses by interrogating the ARP tables of SNMP queryable devices. If a network element connected to the switch was not present in the ARP table of another device, then its IP could not be looked up. This meant the element could not be located within the network model since elements are stored using the IP address as a key. However, the likelihood of devices not being present in the ARP table of SNMP queryable devices proved to be a rare occurrence. At worst the device would appear in my network utility as an orphaned network element which had no link to any other element.

This situation can be remedied by using layer 3 switches, or by querying the ARP table of non-SNMP query able devices in addition to those that are SNMP queryable. This might still miss one or two devices, but the number will be very small if at all.

Another limitation of the network utility is that it provides limited user manipulation of the individual Java3D elements, allowing them simply to be moved around, but not updating the actual model itself. In addition it does not allow the user to update all the fields in the model. Using SNMP set operations it would be possible for some of the devices to be modified using this utility.

5.1 Testing

In order to test the program thoroughly I compared the results it provided to the results from several well respected applications.

Accuracy Testing

To test the project's ability to scan the network and detect devices accurately I compared its output to Nmap, a network scanning tool. In every case was my project able to replicate Nmap's output except for one device, a Speedtouch 716 router. Whilst my network tool would report the DNS name for the router's IP, it would not be able to ping it. Why this happens might be because the Speedtouch router does not respond to the pings which NetworkScan sends via InetAddress. These pings consist of ICMP echo requests (if the privilege can be obtained) or TCP connections on port 7. Nmap uses different types of scans to discover network devices and also has privilege from the system, so this might explain this erroneous result.

In order to determine my project's ability to query an SNMP device with consistent accuracy I used the tool snmpwalk which is able to perform SNMP operations on a host, given the OID, as well as lookup the human readable form of an OID. In all cases the network tool was able to match the output from snmpwalk.

Testing the project's ability to visualise a network was a harder task, and for this I used the trial version of a piece of software called '3Com Network Supervisor'. A network management system, '3Com Network Supervisor' is able to query all devices on the network, scanning for SNMP information and through ARP tables to produce as accurate a network topology diagram as possible. What was clear when using 3Com Network Supervisor was how dependent it was upon SNMP information from the switch and router to construct the network topology. Without the switch, the 3Com utility would not be able to display the connections between individual network elements and the same problem was experienced with my network tool. A network scanning tool will be unable to determine the links between network elements without an SNMP capable device on the network holding this information. When the single SNMP switch was reintroduced, the 3Com utility was able to display individual links. The model I produced accurately correlated with nearly every element the 3Com utility displayed save the Speedtouch Router.

My network utility was not quite accurate as the 3Com utility in that some links between the individual network elements and the switch were not displayed. This was due to the way in which I translated the MAC address information to IP addresses using ARP tables. Whereas the 3Com utility queried the local ARP table, my network utility only queries the ARP tables of SNMP capable devices.

Performance Testing

The network visualisation utility runs smoothly on a Pentium4 2.66Ghz with a fast 3D graphics accelerator, and runs even better on the college lab machines. However, when run across the internet, the network scanning speed slows down considerably.

The table below shows the time for a scan to complete running over different network speeds. Test was performed using a stop watch to keep time and an average of three values was taken:

	Local Network (100 Mbps)	Internet (2048 kbps)
Scanning 64 IPs	29s	58.4s
Scanning 255 IPs	87s	173s

Table 1 – Latency running over different network speeds

The results from the test clearly show the speed increase offered when running the network utility within the Local Network rather than over the Internet. The difference in speed if is almost two fold, but this could be improved by using compression. The simplest way to implement compression over remote connections would be connect to the network to be scanned using a VPN which offers compression. This would reduce the latency since the data being sent is very redundant.

6 Conclusion

The network scanning and visualisation utility developed has been implemented as a single, self-contained application that is cross-platform. The choice of Java aided this goal, and the avoidance of scanning tools pertinent to a particular system (such as Nmap) has made the application more self-contained and portable since it need not rely on a particular system configuration.

If the product used a server-client design pattern, then the server machine could be tailored to the application's requirements, but since this was not the case, portability of the application became more important. An early decision to implement components of the network utility in order of importance led to some functionality being dropped but ensured that a fully functional programme was produced first and foremost.

Implementation of the Common Information Model (CIM) initially required translation of relevant parts of the CIM classes into Java classes. This was relatively straightforward – however what caused a great deal of problems was how to implement associations within CIM. As the number of elements in the network model grew, in some cases a single class being required to hold a single value, the model became difficult to keep track of. As a result of this, every field, method and class was thoroughly commented so that it could be referred to quickly. Having to visualize ten or more classes as well as their associations just to represent a single network element meant that quick access to documentation for each field and association was critical.

The implementation and understanding of SNMP was a very time consuming challenge, requiring the use of two external libraries in addition to handler classes to manage the SNMP functionality. Because of this, handling of Threads, Events and the Observer/Observable model had to be implemented to deal with the delays incurred in invoking and receiving SNMP responses. My knowledge of these was vague to begin with so implementing this one piece of functionality led to a great deal of information being learnt.

The breadth of material covered in order to implement a simple network scanning and visualisation tool is significant, requiring not only code to be written to implement the new functionality, but Nessus server's, SNMP agents, and MySQL databases to be installed and configured.

Dealing with conflicting information from such sources is something I did not implement, assuming instead that any incoming data sources will be accurate. Of course this is not always the case and this project has made me aware of the complexity involved in piecing together a myriad of information sources to form one.

6.1 Future Work

Java Sockets

I initially wished to implement a client-server type design pattern with the client and server communicating via Java sockets. This functionality would allow the server to be able to constantly run on the host machine and maintain a model of the entire network. Multiple clients would be able to connect to the server and manipulate their own views of the model, whilst the model data would remain constantly up to date and inaccessible to any client.

Because of the risk of housing complete SNMP manager capability in a programme, sockets communication would ensure that the client's could not invoke methods remotely on the server.

The downside of sockets however, is that the data is sent in clear text, so the connection between server and client would need to be over SSL before it would be secure enough to use.

Syslog

Syslog functionality was not implemented in the project due mainly to time constraints and also due to a lack of any network device which supported it. It can be added to a Linux installation however, so the potential to try it exists. Also due to its growing use it would be a useful source to include in the project.

Nessus Scanning

Whilst the network utility is able to parse an SQL database of Nessus results it is not able to prompt a Nessus server to obtain new results. This would be an extremely useful feature to have and make the network utility less reliant on external tools having to populate SQL databases for it.

Multiple Users

Along with sockets functionality will come support for multiple users. With connections between client and server secured using SSL, all that is required is individual user authentication for multiple administrators to logon and view a current and real-time representation of their networks.

7 Bibliography

[1] THOMAS D SCHNEIDER - *Representing binding sites in DNA against proteins visually, 1997.*
http://www.ccrnp.ncifcrf.gov/~toms/papers/walker/
[2] DISTRIBUTED MANAGEMENT TASK FORCE. *CIM Specification, 2006.*
http://www.dmtf.org/standards/cim/cim_schema_v212
[3] D. MAURO, K. SCHMIDT, *Essential SNMP 2nd Ed., O'Reilly, 2005.*
[4] B. COLE, R. ECKSTEIN, J. ELLIOT, M. LOY, D. WOOD, *Java Swing 2nd Ed., O'Reilly, 2002.*