dotNetVis: An enhancement and .NET re-implementation of the InetVis Data Visualisation Tool

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

of Rhodes University

Christopher Schwagele

Grahamstown, South Africa
November 2010

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System (1998 version, valid through 2010)

C.2.0 [Computer-communication Networks]: General— Security and protection (e.g., firewalls);

C.2.3 [Computer-communication Networks]: Network Operations—Network monitoring;

I.3.8 [Computer Graphics]: Applications;

C.2.4 [Computer-communication Networks]: Distributed Systems—Client/server;

H.5.2 [Information Interfaces and Presentation (I.7)]: User Interfaces (D.2.2, H.1.2, I.3.6)— Graphical user interfaces (GUI), Interaction styles (e.g., commands, menus, forms, direct manipulation);

C.2.2 [Computer-communication Networks]: Network Protocols—Protocol architecture (OSI model);

General Terms: Security;

Keywords: 3-D (three-dimensional) scatter plot, dotNetVis, internet traffic analysis;

Acknowledgements

Without the expert assistance and guidance provided by my supervisor, Mr. Barry Irwin, the task of completing this thesis would have been insurmountable. I would like to thank Mr. Barry Irwin for supervising my project and keeping me focused towards its completion.

Special thanks must be given to Nicola Tennent for her efforts in proof reading research papers and providing consistent morale support throughout the year. I would also like to thank my family for their continued emotional support during the course of my university career.

Finally, I acknowledge the financial and technical support of this project of Telkom SA, particularly Telkom's Graduate Development Schemes Department, Comverse SA, Stortech, Tellabs, Easttel, Bright Ideas Projects 39, the NRF and THRIP through the Telkom Centre of Excellence at Rhodes University.

Abstract

Persistent and exponential growth in global communication, especially relating to activity on the Internet, continues to drive research and innovation in the field of data visualisation. Traditional approaches used for the capture and analysis of network traffic have become superfluous due to the sheer size of traffic datasets. As a result, there is a high demand for network traffic visualisation systems that offer both efficiency and reliably when working with datasets of a large magnitude.

In 2005, Rhodes Master's student, Jean-Pierre van Riel, succeeded in implementing InetVis, a visualisation tool which, at the time, satisfied the demands of visual traffic monitoring. When significant advancements in computer hardware (observed over the five year span between the original InetVis implementation and the currently enhanced re-implementation) were coupled with the constant rise in network traffic, the InetVis tool suffered major performance issues and could no longer achieve acceptable results.

To resurrect the visualisation tool, we enhanced the underlying structure in a variety of ways. A client server model improves memory and processor usage while at the same time, offers flexibility through modularization, thus allowing easy integration and update capabilities. Multi threading optimizes processing and a redesigned graphical user interface exploits the potential of large screen sizes while maintaining full functionality. These enhancements not only revive the InetVis concept, they offer a means to adapt the efficiency of the tool to allow for its survival within a rapidly-expanding network environment.

Contents

List of Figures

Code Listings

Chapter 1

Introduction

Computer Security is evolving as an increasingly vital component of the Internet. Due to the rapid growth of internet threats and vulnerabilities, an increasing demand for security professionals is emerging. The trouble with security is that any host computer connected to the Internet is vulnerable to an attack and the majority of the users of these hosts are oblivious to the vast amount of malicious traffic their computers receive[11]. Most of these users unfortunately lack the security background to identify malicious activity.

A strong mechanism which can be used to overcome the need for specialised network traffic understanding is data visualization[8]. Visualization techniques are powerful due to their ability to exploit the high-bandwidth visual recognition capabilities of the human eye[19]. It is far easier to identify abnormal network events from a visual model as opposed to traditional techniques which include the sequential analysis of text based log files.

Research in the field of data visualisation propelled Stephen Lau to develop a visualisation tool, named 'The Spinning Cube of Potential Doom', which delved into the visual representation of network traffic in the form of a 3 dimensional cube[15]. Lauś concept was later adapted and enhanced by J.P. van Riel as part of his master's research at Rhodes University in 2005. The resulting tool, labeled InetVis (Internet Visualisation)[21],[23], successfully achieved its goal of providing functionality to carry out visual network security analysis.

1.1 Central Thesis Outline

The advances in computer hardware and the expansion of malicious activity manifesting on the Internet have had a serious impact on the performance of the InetVis tool. As a result, investigation into the enhancement of InetVis resulted in the necessity of an enhanced re-implementation of the tool being identified. Optimization of InetVis will allow for more efficient processing and representation of significantly larger data sets. The user interface is another area which requires an upgrade as it occupies a large portion of desktop real estate, restricting the representation size of the cube. With advancements in design and modeling of graphical user interfaces, a new interface offering identical functionality and improved design would be included.

1.2 Research Methodology

The proposed re-implementation of the InetVis tool will be labeled 'dotNetVis'. This is significant due to the re-implementation being done in the .NET framework. The key differences and upgrades to the InetVis tool are outlined below.

Firstly, the major difference is that dotNetVis is structured around a client-server model implementation. The reason behind this separation is that often, the input into a network traffic visualization tool is provided through a network telescope in the form of capture files. Instead of relocating capture files, the dotNetVis server processes the capture file and transmits only the data required to represent packets to the dotNetVis client, where the visualization will be rendered. This means that processing of the packets is handled on a host separate from the machine used to generate and render the cube, thus providing exclusive processing power to the individual components.

The next upgrade is the advancement away from a static representation of network traffic. By adding the functionality to allow for interactive exploration of the dataset, packet data can be viewed by selecting the packet within the cube. This feature will enable the user of the dotNetVis client to view important information related to any plotted packet within the cube. This will prove useful in the case where the packet being viewed is part of some malicious activity targeting the host subnet.

XNA is used for the implementation of the dotNetVis client as it is more suited to the .NET framework. The XNA client offers the availability of a full screen mode to further extend the maximum data set that can be represented in the cube without obfuscation. A tab-based Windows Forms control is used as a substitute for the controls found in the original InetVis implementation. The new control implementation allows for more efficient and flexible screen real estate utilisation in addition to the improved XNA component.

1.3 Document Overview

In brief summary, this document progresses as follows:

- Background of key concepts vital to understanding the problem domain.
- A review of the InetVis tool which will undergo optimization, including a full specification of its functionality, results and performance issues.
- Development of the dotNetVis tool, outlining enhancements implemented to overcome the InetVis performance drawbacks.
- A conclusion summarising the outcome of the re-implementation in terms of performance contrasts and processing ability.

In chapter 2, background is given into the area of data visualisation and the detection of malicious activity. Chapter 3 lists some of the design considerations that had to be investigated to identify the cause of the performance issues of the InetVis tool. The next three chapters discuss the design and implementation of the dotNetVis application, moving from the server implementation through network communication to implementation of the client application. The final chapter discusses the overall success of the project with reference to usage and performance improvements.

Chapter 2

Background

Before investigation of the InetVis tool can occur, key concepts surrounding the implementation and usage of such tools within a security environment need to be explored. Covering the process of data visualisation, from necessity to malicious activity identification, this chapter serves to provide a concrete foundation on which concepts in later chapters can be discussed. Reference to current visualisation tools, particularly the InetVis tool which is the focus of this project, is used to enforce understanding of the concepts.

2.1 Security and the Internet

The Internet has grown tremendously in the past decade, as illustrated in Figure 2.1. [1] suggests that this trend will continue as global IP traffic is expected to quadruple by 2014. While this growth offers benefits in terms of convergence, quality of life and global development, it also presents an epic challenge to the field of security.

To understand the problems faced by security professionals, it is important to be aware of the vast extents of Internet that require security integration. [21] mentions a global population usage of the Internetin 2005 totalling 14.9%. According to the latest statistics presented by [3], this percentage has risen to 28.7%, nearly doubling in just five years. [3] indicates that population penetration of the internet has increased by 444.8% since 2000.

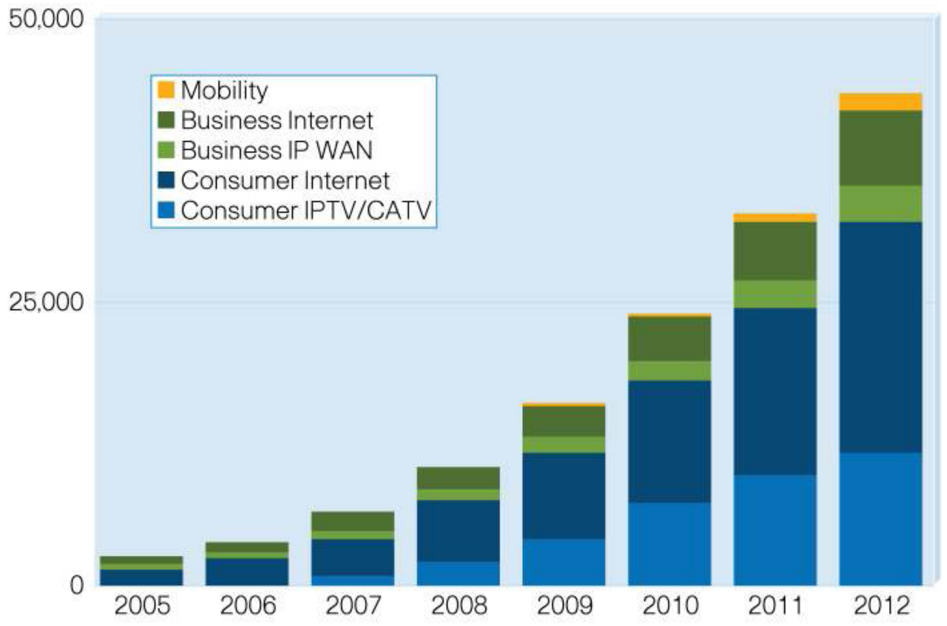


Figure 2.1: Global IP Traffic Growth Forecast

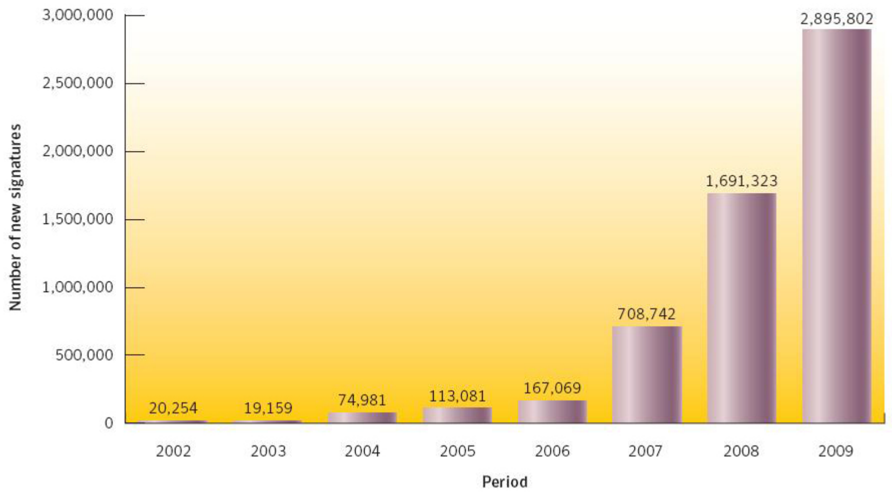


Figure 2.2: New Malicious Code Signatures

It becomes fairly obvious, looking at the growths outlined above, that the existence of malicious activity on the Internet will follow suit. The graph shown in Figure 2.2, provided by Symantec, illustrates the extent to which malicious activity has grown in the past few years. Methods to control the creation and spreading of this malicious activity are non-existent due to the dynamic nature of the Internet itself. As a result, the only countermeasures available must be implemented on the targeted host. This is where security professionals play an important role. Through understanding and inspection of traffic received, a security professional should be able to sanitize incoming traffic and prevent harm to the destination host. A shortage of security professionals prevents this from being a viable solution due to the magnitude of the Internet. Without an in depth understanding of security in relation to computer networks, an individual will find it impossible to identify malicious activity through the use of conventional network monitoring techniques (such as reading log file data in a text editor). Fortunately, research in the field of data visualisation has provided techniques to develop a solution which will address these issues[24],[20].

2.2 Security Data Visualisation

At its heart, security visualisation depends upon graphically presenting security-related data in ways that provide useful and actionable insight[9]. The concept of data visualisation exploits the high-bandwidth visual recognition capabilities of the human eye[9]. This allows for more efficient detection of malicious activity as well as easier pattern and anomaly recognition in comparison to using traditional computing techniques and text-based log files[9]. The Internet has turned the digital world into a dangerous place and security has become a vital component. With continual attacks taking place, the volume of available security data from devices connected to the internet is increasing and this flood of information is obfuscating users perceptions of their security status. Through this confusion, security becomes vulnerable and malicious activity can go unnoticed[9]. Due to the high rate of false positives, traversal of log data becomes tedious. This significantly affects the success rate of a security system[19]. What security data visualisation hopes to provide is a means of aggregating this flood of data into useful information. This is done by drawing graphs, specially crafted for the exact needs of the security system. These graphs provide an extra level of insight and can be used for, but are not limited to, port scans, network sweeps and Denial of Service attack detection[19]. Examples of such graphs are detailed in section 2.3.2 and 2.4.1.

2.3 Network Telescopes

A network telescope is an assigned portion of routed IP address space that is used solely to observe inbound internet traffic[17]. The telescope acts as a dummy host on an isolated network range which means that within the allocated IP address space, no legitimate traffic should exist[13]. This isolated environment allows the telescope to detect and log malicious traffic which could arrive in various flooding of denial-of-service attacks, infection of hosts by Internet worms, and network scanning[17]. Because no legitimate traffic is active on this so called darknet, the majority of packets captured are a result of malicious activity[7]. The typical network telescope will log traffic it receives by storing the packets as capture files. A capture file contains all the information pertinent to each packet received in a given time. The raw packets are aggregated and stored in this format so that a stand-alone tool, such as InetVis, can later read in the capture file and process each raw packet of data so that a more intuitive representation of the network traffic can be generated[19]. The InetVis tool discussed later uses capture files as one of the ways to generate a visualisation.

A class C network telescope is in operation within the Rhodes University IP space and in August 2005, 867 085 packets were captured and used as a testing input for the InetVis tool[22]. A screenshot of the InetVis tool displaying this dump file can be seen in Figure 2.3. The benefits of using a visualisation tool to view captured packets should be apparent in the given screen capture. The horizontal lines seen in the lower section of Figure 2.3 are caused by network probes which are used to scan an address range to locate vulnerable hosts which will undergo further scans upon discovery. Because the telescope is passive in nature [10], no nodes exist within the address range[22].

2.3.1 Areas of high traffic

In Figure 2.4, an orthographic view along the blue x-axis (representing the network telescope address space), groupings of plotted traffic become apparent. This indicates that those particular address ranges are a rich source of malicious traffic and as such, precautions can then be put into place to protect against those hostile ranges.

2.3.2 Malicious signatures

With a visualization of such large magnitude, some common signatures begin to emerge. As seen in Figure 2.5, anomalous diagonals are apparent. An orthographic view along the z-axis shows targeted ports as well as targeted IP Addresses. The diagonal lines that are visible in Figure 2.5 are caused by a scan known as a barber pole [15]. This scan type

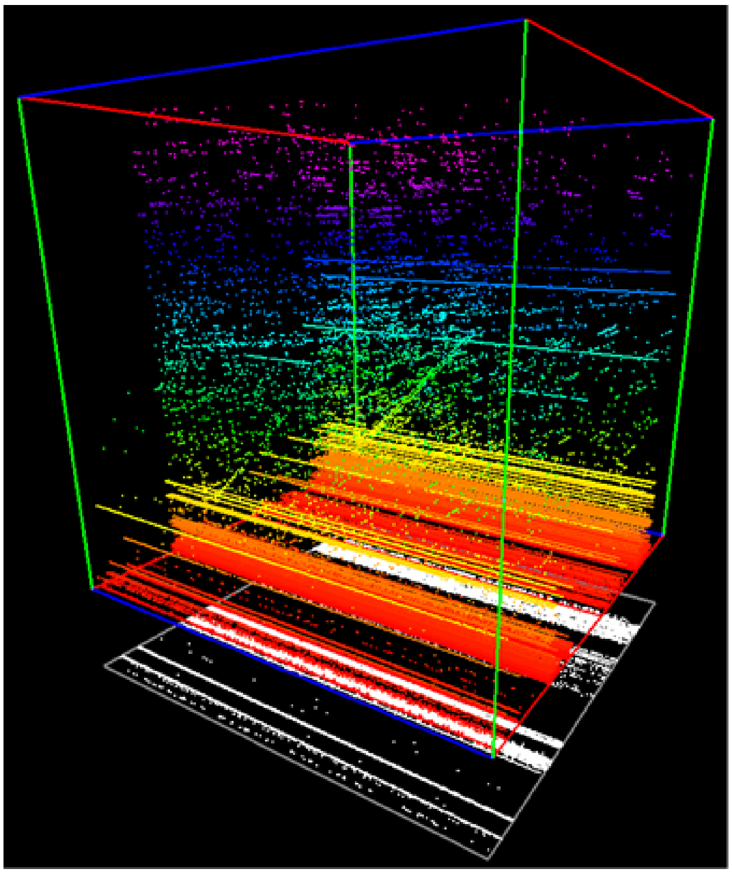


Figure 2.3: InetVis displaying telescope traffic from August 2005

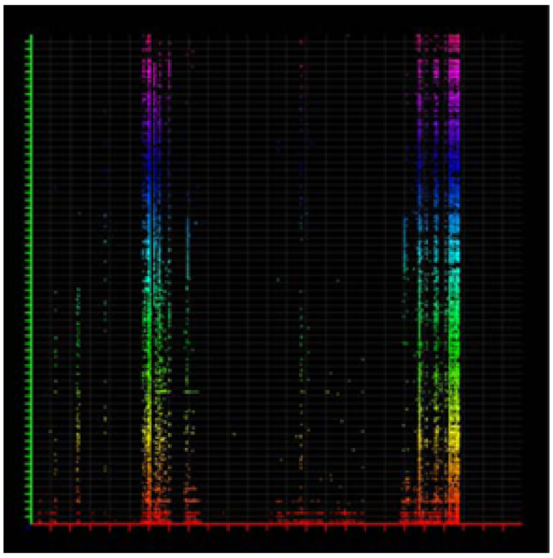


Figure 2.4: Groupings form along the IP source range highlighting hostile ranges

is discussed in more detail later on. Two other scan types are visible as well. These are shown in Figure 2.6 and Figure 2.7 respectively. Figure 2.6 represents a step scan whilst

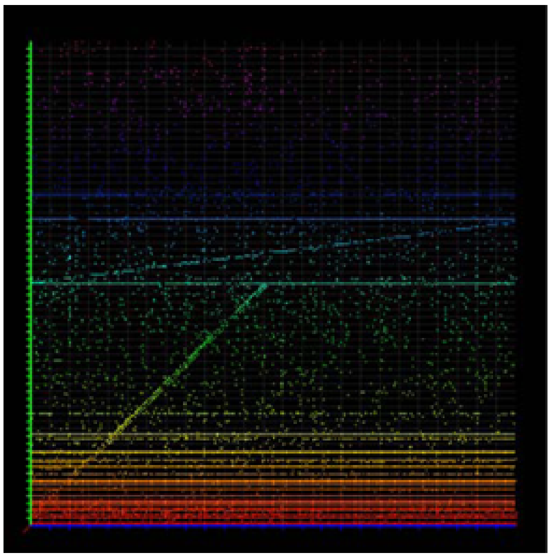


Figure 2.5: Anomalous diagonals

Figure 2.7 represents a creepy crawly scan[22].

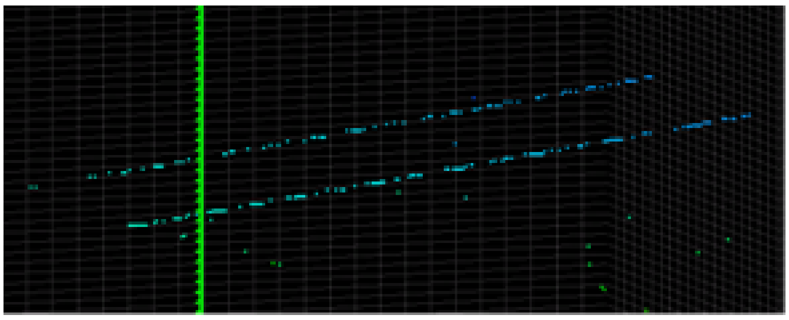


Figure 2.6: Step scan signatures

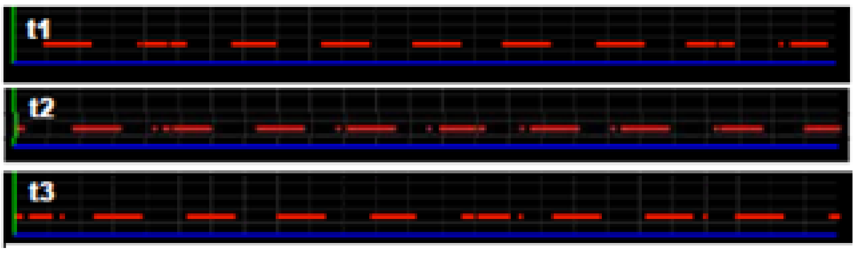


Figure 2.7: Creepy crawly scan shown at different intervals

A step scan divides a target network range into blocks. It starts at the first block and scans a specific port. When it reaches the end of the block, it selects a new destination port number and scans the following block. It repeats this until all network addresses within the network range it has set out to scan, have been scanned [22].

A creepy crawly attack works in a similar way. Its goal is to scan a designated IP range. It divides the range up into segments and then scans a subset of these segments simultaneously. This gives the dashed line effect shown in Figure 2.7. The three different time intervals, t1, t2 and t3, all show the dashed line pattern and when the traffic is viewed over the duration of the scan, a solid line will be noticeable [22].

2.4 InetVis

Having looked at network telescopes and the strength of data visualisation, it is time to explore the InetVis tool in detail. InetVis is a 3D animated scatter plot network traffic visualisation tool. The function of the InetVis tool is to transfer an input of IP layer packets into a three dimensional cube visualisation which plots packets according to source and destination IP addresses as well as the destination port number. Figure 6 shows an InetVis visualisation of an NMap gridsweep scan with decoys. The lines that can be seen running parallel to the blue x-axis are scans that are taking place on common listening ports across a subset of the home IP Address space. As can be seen, there are four identical bands coming from four different source IP addresses. This is because three of the four scans are spoofed decoys. Only one of the scans is legitimate. The fading effect that is visible is a feature of the InetVis tool which fades packets as they age. The plotted packets are coloured based on their destination port number [22].

2.4.1 Concept

InetVis is based on a similar implementation by Stephen Lau which is called The Spinning Cube of Potential Doom [22]. Laus primary reason for creating the cube was to provide a means of education to those who are not experts in the field of computer security [15]. Lau wanted to create an application that could highlight the overall extent of malicious traffic on the Internet.

It is reasonable to note that the majority of computer users are blissfully unaware of the amount of malicious traffic circulating the internet on a daily basis. This lack of awareness is an issue that needs to be addressed in the field of computer security and Laus cube has made an attempt at satisfying that need. Through visualisation, data can be represented in such a way that merely observing the visualisation will allow a user to identify malicious activity [16].

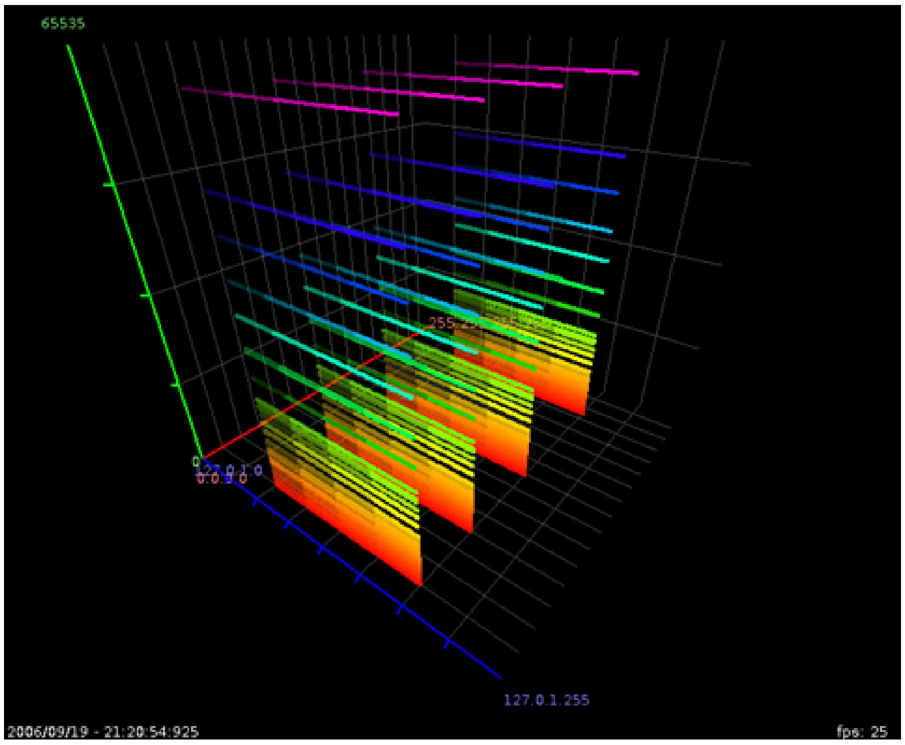


Figure 2.8: InetVis displaying an NMap generated gridsweep with decoys

Laus cube leverages off the Bro IDS [7] which monitors TCP connections and logs source and destination IPs as well as port numbers. Laus cube uses these logs as an input to plot each packet according to the following layout: The X-axis is used to plot the home network (which the IDS is monitoring). This is typically a subnet in an organisation. The Z-axis represents all possible IP address space (from 0.0.0.0 to 223.255.255.255 which excludes multicast traffic). The Y-access represents the port number. Using these axes, certain types of malicious activity can be identified. Various visual signatures are discussed below.

Port Scan

A port scan occurs on a single host. Attackers can use port scans to acquire critical information about a hosts machine [18]. A port scan will usually report a list of open ports on the targeted machine. These ports are essentially tunnels into the secure host and with appropriate tools some ports can be used to gain unauthorised access into the host machine. A port scan can be identified by observing the traffic between a monitored network node and the source IP address of the packets that that node is receiving. If a single source IP address is seen communicating to multiple ports on a host machine, then it is highly probable that a port scan is taking place.

There are many tools that can be used to carry out a port scan and some of the more clever tools attempt to be as discreet as possible [18]. This is achieved by varying the delay between packets sent to the host. This makes it difficult to see the overall port scan signature as packets do not arrive uniformly. There are other methods that can be used to hide a port scan as well but that is beyond the scope of this thesis. A typical port scan is shown in Figure 2.9 which is a screenshot of a visualisation achieved using Lau's cube. A visible horizontal line is formed which indicates that a single source IP address has sent traffic to a multitude of port numbers on a single host.

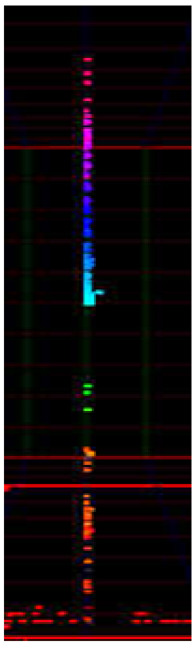


Figure 2.9: A port scan signature with Laus Cube

Barber Pole Scan

Barber pole scans occur across an array of hosts and they are harder to trace due to the fact that they vary their destination port number and destination host machine simultaneously. A text based intrusion detection system will have difficulty in identifying this kind of an attack [19], but when represented as a visualisation, such an attack is clearly visible. A barber pole signature is shown in Figure 2.10 and it is identified by distinct diagonal lines.

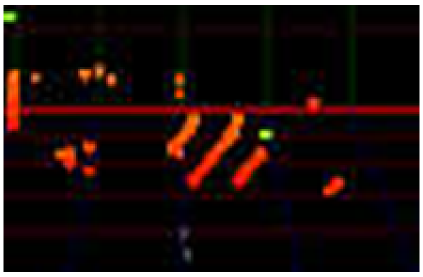


Figure 2.10: A Barber Pole signature with Laus Cube

Tools that are able to perform these scans are sophisticated in a sense that they have the ability to skip IP addresses and port numbers as well as scan more than one port on a target host [15]. This flexibility decreases the detection rate as fuzzy patterns become more prominent.

Lawnmower Scan

A lawnmower attack occurs across a wide range of contiguous ports and hosts. There is no discretion when one of these scans takes place. The tool used to initiate the scan will typically proceed to sequentially scan an address range by targeting a specific port identical to a port scan algorithm. Once the range has been traversed, the destination port is changed and the scan repeats [15]. As shown in Figure 2.11, the lawnmower scan signature is highly visible in a visualisation environment. The only difficulty in detecting a lawnmower scan is that the time it takes to execute is variable, as with any other scan. Because of this, the visualisation may decay before any visual signatures can be inferred.

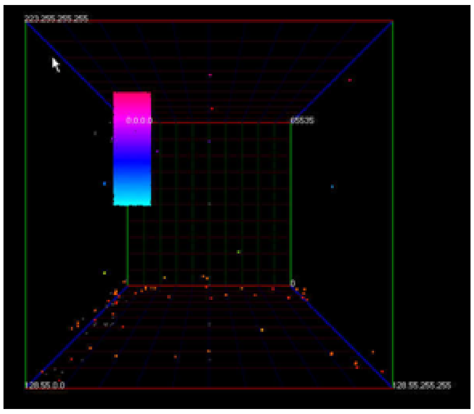


Figure 2.11: A Lawnmower signature with Laus cube

2.4.2 Implementation

Although InetVis is based on Laus cube, it differs in many regards. InetVis was created as an extension of Laus work and it includes useful features such as variable playback rate, an adjustable time window and filtering via the Berkley Packet Filtering method [22]. InetVis uses a network telescope as its primary source of data. The capture files which are created by the telescope can be visualised within InetVis through an animated 3D scatter plot. Laus cube uses the Bro IDS as its source of network traffic input [2]. InetVis has been tested on a class C network telescope set up in the Rhodes University IP space and it passively monitors traffic received within this space. InetVis is not limited to the input of historic traffic (in a sense that the traffic is not monitored in real time) as it has the ability to monitor live network traffic as well [22]. This is done by capturing traffic received on the local network adapter of the machine on which the InetVis application is executed.

Plotting Scheme

Figure 2.8 illustrates the plotting scheme of InetVis. The purpose of the cube is to visualise packets captured within a predefined network range, having received them from any source IP address. Packet inputs read into the InetVis tool for processing are handled by libpcap, a library used for the low level capturing of packets in a network environment [5]. InetVis plots traffic found at the IP layer which includes TCP and UDP as well as ICMP traffic. The latter type has no port dimension so packets of this type are plotted on a flat ICMP plane under the InetVis cube. TCP and UDP packets are plotted within the 3D cube environment. The location of a plotted point in the cube depends on three variables. The x-axis of the cube is mapped to the IP range of the home network (i.e., the destination IP address of the captured packet). The z-axis of the cube is used to map the source IP address of each packet. The y-axis maps the destination port range of the packets captured; provided they are of the type UDP or TCP. InetVis colours points according to the destination port of a packet. A useful feature is that points can be coloured based on other dimensions, such as packet size, source port numbers or protocol type [22]. The size of each plotted point can be changed based on the visualisation needs. This allows for efficient visualisation of large and small sets of network traffic data.

Features

Perhaps the most useful feature of InetVis is the ability to visualise capture files at flexible playback rates [22]. This enables users to navigate through a capture stream in an efficient manner, allowing them to skip periods of non-interest as well as playback an important time slice at sufficiently slow speeds. InetVis allows replay rates to range between 0.001 times (one millisecond per second) adn 86400 times (one day per second) normal speed [22].

The concept of a variable time window is also included and its usage is first seen in VisFlow-Connect [25]. The concept of a time window allows an event to exist for a specified length of time after it has occurred. This is a very important feature for a tool like InetVis as scans take place over varying time periods. Some scans progress over the course of a month, whilst others may execute in a matter of seconds. Due to this flexibility, a time window is considered a valuable component. The OpenGL cube which is generated by InetVis would be fairly useless if there were no way to navigate the 3D environment. Following in the line of the Space Shield visualisation [10], InetVis adopts a simple 3D navigation implementation which allows the user to navigate the 3D environment by using the mouse to zoom, rotate and move.

The OpenGL component of InetVis also supports both orthographic and perspective projection modes. Perspective viewing allows for a sense of depth and a more realistic feel to the cube. The orthographic projection mode allows for a more accurate and meaningful geometric representation. This means that any line going in the same direction in a given dimension will be parallel to its counterpart. This view is more appropriate for making assumptions based on the visual representation of the traffic.

Apart from the actual 3D navigation and time window components, support for Berkley Packet Filter (BPF) expressions is also included. These BPF expressions allow filtering of the captured packets which decongests the amount of data displayed simultaneously. This is useful for when monitoring of a specific host is required. Reference axes are included as optional displays which are designed to assist the user in dividing the cube into subsections. These options, along with all other mentioned features are available through the control panel and plotter settings windows of the InetVis tool.

Performance Scalability

As shown above in Figure 2.3, a test involving all the traffic captured on a class C network telescope over August 2005 (some 857085 packets) was used to show that InetVis can display large data sets correctly. Tests detailed in [21] reveal at maximum replay speed (86400 times), an approximate 30000 packets were displayed each second [22]. This was achieved on a machine with just 1 GB of RAM and a single core 3.0GHz Pentium IV processor. According to van Riel [22], a framerate of 25 frames per second was achieved until approximately 450000 events had been plotted. InetVis could not maintain the 86400x speedup after this much data was being displayed [22].

2.5 Chapter Summary

The central concept surrounding the necessity of visualisation tools has been defined and it is evident that the trends followed by network growth and malicious activity are pushing for the implementation of innovative visualisation tools. InetVis has been discussed and it becomes evident, through analysis of the performance of InetVis in lieu of large dataset representation, that various enhancements need to be implemented to optimise the underlying visualisation tool.

Chapter 3

Design Considerations

This chapter serves to identify general design considerations that must be explored before the re-implmentation of InetVis can ensue. These design considerations are relevant to the overall structure and environment of the resulting implementation. Technological abilities are discussed, and the chosen architecture is then outlined.

3.1 Hardware Advancements

The InetVis tool has proven to be successful in providing a means to solve the problem of malicious activity identification. The performance issues plaguing the implementation are merely the result of technological advances, thus the underlying concept portrayed in InetVis suffers no loss in its usefulness. In fact, the technological advancement, which is the reason behind the performance issues faced by InetVis, presents the opportunity to solve the underlying problems. Processor technologies offer a high degree of support for multithreaded applications which can be used to optimise data processing of large datasets. A substantial increase in memory capacity offers faster storage and manipulation of these large datasets. By combining the available memory capacity and processing power with optimized techniques tailored for large dataset manipulation, the performance barriers evident in InetVis can be overcome.

The other issue surrounding the InetVis implementation is the lack of efficient desktop space usage. Larger datasets will need to occupy a larger screen space. The size of monitors available today differ significantly from what was on offer five years ago. By extending the InetVis tool to include fullscreen support, very large datasets can be represented visually without the issue of data overlap (When the scale required for visulisation exceeds the number of pixels available).

3.2 Development Environment

As a further extension to the InetVis tool, flexibility, in terms of future extension implementation, will be a key focus area in the new design. The InetVis tool overcomes many of the problems faced in the process of identifying malicious network activity but it lacks in certain areas. The problem is that the InetVis implementation is not easily extendible due to its underlying architecture. To offer the flexibility of modularized development, as well as trivial integration of enhancements without a tradeoff in performance, the .NET framework was chosen for its re-implementation.

While cross platform development is not natively supported by the .NET framework, the flexibility offered is far better than can be achieved in C++. Development of applications is rapid and graphical interfaces are easily implemented. This will aid in the future integration of enhancements to the re-implementation.

In response to the necessity of an enhanced design of the graphical user interface (GUI), C# offers a flexible and rapid approach to GUI construction, as well as a rich collection of professional GUI controls. In contrast to C++, the .NET framework offers substantial support and a large collection of custom libraries, enforcing the rapid development approach.

Furthermore, the rendering component used for the visualisation is implemented in XNA, a natively supported 3-D engine for .NET. A wrapper class allowing openGL development in .NET was also considered, however it lacked the full functionality required for an exact re-implementation. The XNA game development library was used instead and its integration with the system was easily achieved.

3.3 Underlying System Model

To successfully offer the flexibility to allow future integration, the underlying model of InetVis required revision. Firstly, the InetVis application comprises two general components. Input is gathered from a capture device and an output visualisation is generated using this data. By separating these two components, a more distinct process is formulated. Input is handled by one component and output is handled by another. Due to the resulting independence of the components in terms of processing separation, the model can be extended further to include a network component following the client server model approach.

This model is favored for a number of reasons in the context of dotNetVis. The main advantage of adopting such a model is that the processing requirements and memory management of a complex system can be distributed amongst the independent hosts making use of that system. In the case of InetVis, data flow is relatively straight forward in terms of dissection. Data are read in from a capture file or from the local host. The data are then processed and filtered based on intrinsic network parameters. Following the processing, the relevant data are then sent to a rendering module and an output visualisation is generated. Through the identifiable modularization of the data flow from source to sink modules, it is evident that the application can be broken into two independent data processing components. By isolating these two components, namely data aggregation and data visualisation, on two separate machines linked by a valid Ethernet connection, more processing power can be provided and more efficient processing techniques can be implemented to allow for a more flexible system.

3.3.1 Structure

The capturing of network traffic is handled by a third party application that utilizes a darknet[1]. The server component of dotNetVis is used only to process network traffic, extracted either from the capture file provided by the third party application such as tcpdump [4] or wireshark [6], or captured from a live traffic stream on the local host, provided by the local network adapter. The server component will evaluate each network

[1]A darknet is equivalent to a network telescope. See the section 'Network Telescopes' in Chapter 2 for a definition

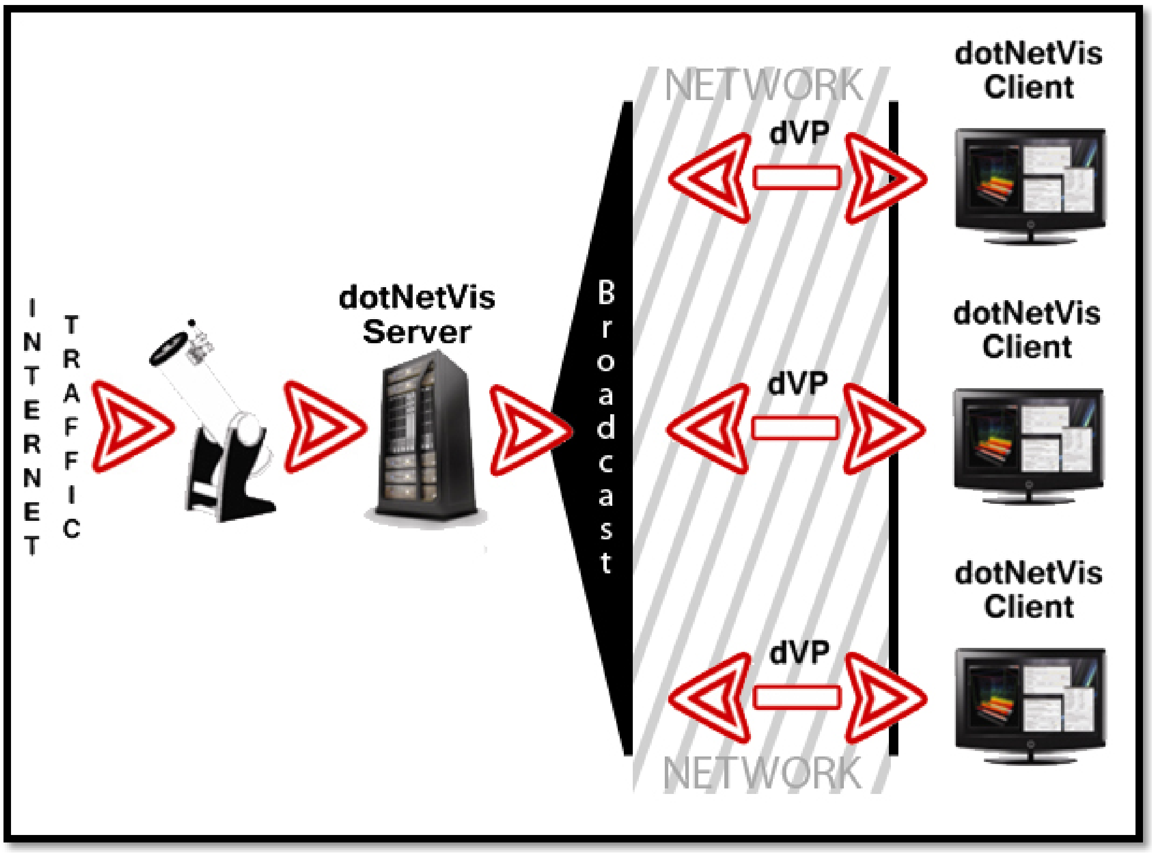


Figure 3.1: Representation of the client server model within the dotNetVis system

packet it receives and for each IP packet identified, it will transmit the pertinent data required to plot the packet, to the client application.

Communication between client and server applications is handled by the dotNetVis Protocol (dVP), a standard specifically engineered to handle requests and responses between dotNetVis client and server instances provided that there is a standard Ethernet link between the two components.

The dVP API is then used by the requesting[2] client instance to gather the server-processed data. These data are then stored in an ordered collection based on the original IP packet's time stamp. A manager object is used to move subsets of the stored data from the client's dVP receiver library to the rendering component, as requested by the client application's user. The rendering component will then use the subset of data it receives to generate a

[2]A client component must request streaming of data from the server by specifying the data source (capture file or live monitoring). The server thus operates on client demand rather than on supply from a capture device. This gives control to the user of the client instance

scatter plot visualisation to display the network packets belonging to that subset.

3.3.2 Implications

The client server model approach will allow for easy future implementations (in terms of integration) of either the dotNetVis client or the dotNetVis server component. Due to the server component being a stand alone application, any server implementation which makes use of the dotNetVis Protocol (dVP) library's functionality will be able to communicate with the server instance through the use of the available dVP methods. Full details regarding the library are given, and the dVP methods are detailed, in a later section of this paper. The dVP library allows for effortless integration and modularized development of future client and server applications, thus allowing flexibility to create specialized server applications which can independently handle data collection; as well as specialized client rendering applications which can, independent of the server application, run on any platform or technology which supports the dVP library. This flexibility on both sides allows the dotNetVis tool to be tailored to application-specific needs.

3.4 Chapter Summary

The client server model has been identified as a highly adaptive approach to modularization of the dotNetVis system. By implementing this model, performance benefits are increased beyond the potential of parallelism techniques. With the structure defined, the next few chapters will discuss development of the separate components in detail.

Chapter 4

Server Application Development

The dotNetVis server has been implemented as a console application using Microsoft's C# development language. Due to the simplicity of the server application in terms of user input, a graphical user interface would be superfluous in terms of necessity and hardware utilization. The only input required at the initialization stage of the server application is textual. This input defines the operation of the server and it should be provided by the user at one of two stages.

The user can specify the input settings via command-line parameters when starting the server application from a command-line shell. The command line entry will take the form 'dVServer x' where 'x' can be one or more of the parameters outlined below.

- Server port

  The port on which the server will broadcast data and accept client connections can be set using '-p portNumber', where 'portNumber' represents the number of the port on which the user requires broadcasting to take place. A value outside of the allowed port range, or a port that is already in use, will not be accepted and the server will indicate to the user that a different port number needs to be specified. If a port is not manually set by the user, the default port value[1] will be used.

[1]The default port value is defined in the server source code and is set to port 31337. The default value cannot be altered in the current version of dotNetVis

- Server Address

  This value is used to specify the IP Address that will be used by clients to connect to the host machine on which the server application is executing. The argument used to specify a server address is of the form '-a ipNum' where 'ipNum' is used to specify an IPv4-standard IP Address in dot-decimal notation[2].

  The server will parse the specified address and it will attempt to match the address to an available networking interface on the local host machine. If there are no matches for the specified IP Address, the user will be prompted to specify a different IP Address. If the address is not specified manually, the server will use a simple domain name resolution service to list the IP Addresses of all the valid network interfaces available on the local host. The user must then select an address to use for client-server communication from the list provided.

- Default capture file

  The dotNetVis server makes use of a default capture file directory to keep track of, and retrieve capture files. The server also maintains a default capture file name which is used when a dotNetVis client makes a request for a capture file. This removes the necessity of specifying which capture file to process with every capture file request.

  To change the default capture file directory, the argument '-cd directory' can be used. The 'directory' parameter must represent the location (relative or absolute) of the newly selected default directory in which capture files will be located. If the directory does not exist, the server will request that the user specify a valid directory. If no directory is specified, the default location[3] will be used. It is important to note that once the dotNetVis server is running, the default directory for capture files cannot be changed.

  The name of the default capture file is used when a capture file request is received. The server will always process the capture file specified by the default name. To

[2]An IPv6 IP Address cannot be specified using command-line parameters in the current version of dotNetVis. The alternative method to select a network adapter must be used in the case where an IPv6 address is required

[3]The default directory (relative address) is '.
cap_files
'

change the default name on startup, the argument '-cf capFile' can be used. The parameter 'capFile' specifies the name of the default capture file to be used. If the file does not exist in the default directory, the server will prompt the user to enter a valid capture file name.

The dotNetVis client application has primary control over the default capture file value. A client application can request a listing of capture files available in the server's default capture file directory. A user on the client side can then select a capture file for visualisation from the list of available capture files. The client will send a request to the server which will change the default capture file to the one chosen by the client application's user. When a capture file request is received from the client application, the newly specified default capture file will be processed.

The alternative method available for configuring the server is on server request. Once started, the server will begin initialization and it will use default values for all the options outlined above, except for the server address. The user will have to select an IP Address from the list of addresses displayed. Once specified, the server will be ready to accept clients on its default port.

The first method of input offers two advantages to the user. Firstly, startup and initialization of the server application can be automated using a script or batch file. Secondly, the server can be fine-tuned to meet the requirements of the user.

4.1 Application Flow

When the application is started, a console window will appear and the server will enter a waiting state, where it will listen on the specified port for a client connection request.

When a client connects, the server will listen for client requests. Requests that alter server settings will be processed and a response stating the success of the alteration will be sent to the client. When a request to process a capture file is received, the server will determine the network source that the client has chosen and it will initialize the capture device accordingly. The request will trigger the creation of a new packet-processing thread which will handle the processing of network traffic on the specified capture device. Once the capture is completed, the packet-processing thread will terminate. The server will indicate to the client that processing is complete and the client will disconnect. The server will revert to its waiting state until a client connection request is received. This flow is shown in Figure 4.1.

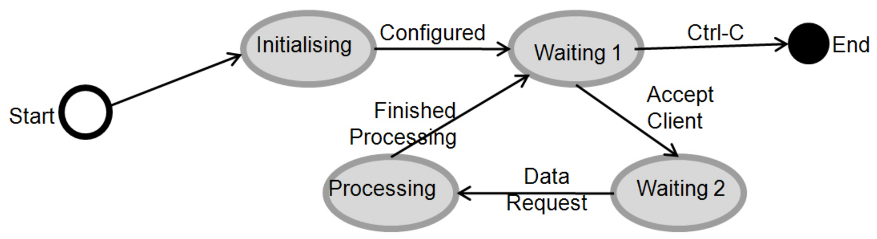


Figure 4.1: Anomalous diagonals

4.2 Capturing Packets

The implementation of the network monitoring and packet capturing component of the dotNetVis server follows the same structure as that of the WinPcap/libpcap component used in the InetVis application. The WinPcap library is a port of the libpcap library, which was developed for the purpose of low-level packet capture, capture file reading and capture file writing.

Tamir Gal developed a packet capture framework for the .NET environment called SharpPcap. The SharpPcap library is an adaptation of the WinPcap library and its sole purpose is to provide an API for capturing, injecting, analyzing and building network packets within the .NET framework.

The dotNetVis server makes use of the SharpPcap library [12] to inject processed network traffic into the visualisation component of the dotNetVis client. There are two ways in which the SharpPcap library can provide this functionality. The library can either be used to monitor live network traffic on a local network adapter, or it can be used to examine and interpret the contents of a capture file, the contents of which represents a collection of past network traffic in the form of packet events.

The first step in the process of capturing network traffic, regardless of the source, is to select an appropriate packet capture device. The device will either represent a physical network adapter or it will take the form of an off-line packet capture device, in which case it would represent a capture file located on the system's hard disk.

4.2.1 Monitoring traffic on a chosen device

Once a device has been specified, the SharpPcap library can then open the device and begin the capture process. The flexibility of the library provides two functions that can be used to start capturing packets on the device. The first variant will create a separate thread which will continue capturing packets on the device until it is explicitly stopped by the parent thread. The other variant requires an integer parameter to be specified, which indicates the number of packets that the device should process before it terminates the capture process. Capturing packets on a separate thread will allow the parent thread to continue execution. The latter variant will block execution of the main thread until the specified number of packets is processed.

An implementation of either of the methods to capture packets will allow two options for reading in individual packets. The first option is to register an event handler on the capture device. The 'PcapOnPacketArrival' event will trigger on every packet that is captured by the device. This event-driven approach will give control over the processing of packets to the capture device since packets are processed on supply.

The alternative method, namely 'PcapGetNextPacket', will return the next available packet in the capture device's data stream. By implementing this approach, direct control over the processing of packets is maintained in the user application. The drawback however is that an explicit call to receive a packet is required, thus introducing the issue of buffer overflow in the device's capture buffer. For example, if the device is monitoring live traffic (let's say on average it receives 10 packets per second) and the call to retrieve a packet from the device for processing occurs once every second (due to complex packet processing), then the device's buffer will increase by 9 packets every second, inevitably resulting in buffer overflow. In this scenario, the explicit retrieval of packets is not a viable solution. Of course, the complex packet processing in this scenario is the bottleneck and it would make sense to use a multi-threaded approach to solve the issue, but the scenario demonstrates the issue with the explicit retrieval of packets. The dotNetVis server application uses the explicit function call variant to parse capture files as the packets don't require real time processing.

Another useful feature that dotNetVis takes advantage of is the packet filter that can be applied to a capture device. Since the dotNetVis client application is designed to render only IP traffic, efficiency can be extended at the server level by applying an IP filter to the capture device. The following code listing shows how the dotNetVis server filters traffic and only captures packets that conform to the IPv4 standard:

Listing 4.1: Filtering packets for IPv4

```
string filter = "ip";
//Associate the filter with this capture
device.PcapSetFilter( filter );
```

By applying such a filter, packets that are not relevant to the visualisation component can be ignored and processing of these packets can be skipped. All packets that are successfully retrieved on the capture device will undergo processing once they have been cast as SharpPcap-defined Packet objects.

4.2.2 The Packet Processing Process

At this point in the dotNetVis server's execution, IPv4 packets are being received piecemeal by the capture device. The packet object then undergoes processing so that the pertinent information required to represent the packet on the client side can be extracted. The most important value contained within the packet is the packet's identifier. This value is merely the index of the captured packet, obtained from a counter that increments with each processed packet[4]. The rest of the information is extrapolated from the packet object. The arrival time of the packet is represented by the .NET System.DateTime object. The source and destination addresses as well as the destination port of every packet

[4]The captured packets are a subset of the capture file due to the filtering that occurs once the device has been opened. To clarify, if a packets identifier value is 68, then that packet is the 68th packet in the subset of packets that meet the criteria of the applied packet filter (e.g., in the subset of IPv4 packets). The identifier does not represent the packets index in the capture file

are also extracted. This 5-tuple packet representation is then transmitted over the network via the dVP transmitter library.

This process is repeated for each packet that is captured by the capture device until all packets have been processed, or until the device terminates capture. Differentiation between packet types and the extraction of pertinent data is illustrated in the following code listing. The last line of code shows the usage of the dVP API to transmit the 5-tuple:

Listing 4.2: Packet processor: identifying and transmitting packets

```
IpPacket ipPacket = (IpPacket.GetEncapsulated(Packet.ParsePacket(packet)));
if (ipPacket != null)
{
  switch (ipPacket.Protocol)
  {
    case IPProtocolType.TCP:
    {
      TcpPacket tcpPacket = (TcpPacket)ipPacket.PayloadPacket;
      //*** 5-tuple qualification: ***
      long id = ++count;
      DateTime time = ipPacket.Timeval.Date;
      IPAddress srcAddress = ipPacket.SourceAddress;
      IPAddress dstAddress = ipPacket.DestinationAddress;
      ushort dstPort = tcpPacket.DestinationPort;

      //use dVP library to send the tuple
      Transmitter.Transmit(0,                    //standard tuple transmission
                           Transmitter.version,
                           id,
                           srcAddress.ToString(),
                           dstAddress.ToString(),
                           dstPort,
                           time);
      break;
    }
```

4.3 Chapter Summary

Having completed this chapter, the process of input capture and transmission to the client should be well defined. With complete understanding of the server component, the next component can now be investigated.

Chapter 5

Client Server Communication

The client server model that the dotNetVis system has adopted, primarily offers flexibility in terms of independent client and server implementation. As mentioned previously, this allows for the development of flexible, platform independent variations on both the server and client components, allowing for customization of the tool without requiring a complete re-implementation of the system.

Despite the modularization and the independence of the components, both client and server applications, regardless of the underlying platform, design and hardware, must make use of one very important component: the dotNetVis protocol API. The necessity of this interface was realized when a flexible standard of communication could not be identified for the purpose of dotNetVis client-server interaction. As a result, the dVP library was designed and an API that encapsulates the library was built.

5.1 The dotNetVis Protocol

The dVP is an application layer protocol ([14]) which uses TCP as its underlying transport layer protocol to ensure guaranteed dVP packet delivery. The dVP is designed to be independent of the transport layer protocol and can be extended with additional capabilities. Implementation of the dVP is still in its infancy and thus the benefits it currently offers to the dotNetVis system are primitive in contrast to its future potential.

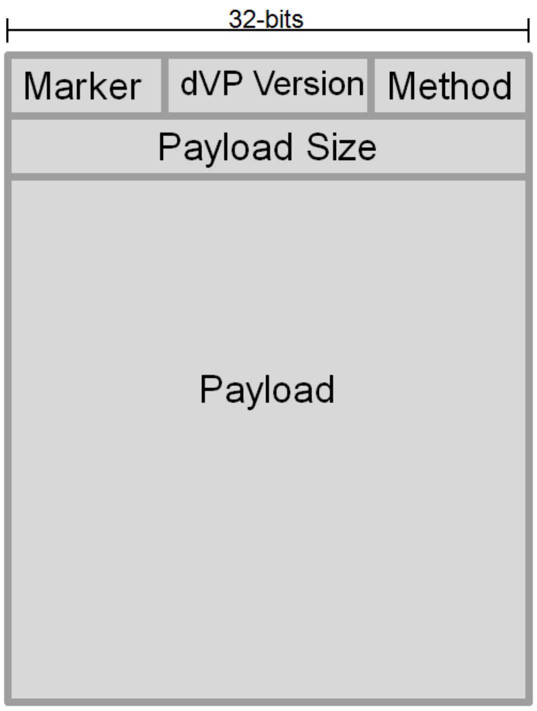


Figure 5.1: dVP Packet Structure

Figure TODO shows the basic structure of a dVP packet. The size of a dVP packet header is 64 bits in total. The two message types used by the dVP are request messages and response messages. The payload size varies according to the message type being used. The header fields for all dVP packets are outlined below:

- Marker

  The marker field is a single byte which is used as a flag to indicate the arrival of a valid dotNetVis packet at the receiver. The same marker must appear at the end of all dotNetVis packets to validate their authenticity. If a packet does not satisfy both conditions, it will be discarded. This will prevent the receiver from attempting to process invalid packets, to a certain extent, the exception being when an invalid packet happens to start with an identical byte. The default marker value is 0xFF and it is defined in the dVP library.

- Version

The first two bytes of the version field indicate the dVP's major version number and the second two bytes indicate the dVP's minor version number. This is to ensure compatibility between client and server dVP libraries in the case of a version change.

- Method

  There are a number of methods available for use by both client and server applications. The following table summarizes the different method types:

0x00	Indicates that the payload represents a processed IP packet which must be added to the client application's list of scatter plot points.
0x01	Indicates that the message is a request for the server application to begin processing the default capture file. The payload is empty.
0x02	Indicates that the server application has completed processing packets from the specified capture device. The payload is empty.

- Payload Size This token indicates the size of the payload data in bytes, represented as an integer value. It is used to indicate how many bytes should be read by the receiver before the marker byte is checked.

All dVP packets are crafted and managed within the dVP library. The dVP API defines a multitude of methods that can be used to provide a communications infrastructure within the dotNetVis system. Full library code is provided in Appendix A. The library offers reliable connection management between client and server applications through the use of buffered data streams, single sockets[1], and TCP listener objects, all of which are defined in the standard Windows System library.

The dVP library is packaged as a dynamic link library (.dll) file which is theoretically supported across multiple platforms[2]. By encapsulating the library in an assembly, the difficulty when integrating dVP functionality into a client or server re-implementation is kept to a minimum. This approach further enforces the independence of the client and

[1]Single socket implies that the utilisation of sockets on any dotNetVis application instance, be it server or client, will be limited to a single socket object at any given time. Concurrent socket objects are not allowed

[2]For Linux use: ensure that the dVP.dll as well as the windows System.dll libraries are registered in the Global Assembly Cache (GAC). Compile the dotNetVis source files, ensuring reference to the libraries is specified, by using the mono compiler

server applications.

5.2 dVP API Usage

While many functions are defined in the dVP library, the external use of these functions is limited through an abstraction of the dVP's full functionality. The interaction between the server and client components of the dotNetVis system and the dVP API requires careful explanation to ensure that the procedure of communication is well defined.

5.2.1 Server Communication

The server component of dotNetVis will have the ability to collect data in some specified way that is completely up to the developer of the server component. It is then the job of the dVP to provide a means of transport to deliver the data to the client application.

The functionality exploited by the server application is merely to receive client connections and requests, stream input data, and send status responses.

Client Connections

The server will enter a waiting state at various points in its life cycle. Client instances establish non-persistent connections based on distinct requests. Once the server has been configured, it will create a dVP.Transmitter object which requires the server address and port number on which transmission will take place as initialization parameters. During initialization, the transmitter object will initialize a new TCP listener object for TCP communication.

The server will enter its waiting state by calling the dVP API's static 'wait()' method. The transmitter will listen on the port until a valid TCP connection request is received. A network stream will be created and the server will wait for requests from the connected client by listening for valid dVP packets on the designated socket.

Data Request

When a request for data streaming is received, the transmitter will shift the server into a processing state. The data request will be used to identify the requested input source, either a capture file or a live network interface, and the server will initiate the packet processing thread accordingly.

Data Transmission

During the packet processing process, packets will need to be transmitted to the client application. The dVP API defines various 'transmit()' methods that can be used for this purpose. The packet processor extracts 5-tuple collections from each packet requiring transmission. This tuple is passed to the transmitter via the appropriate tranmit() method.

The transmitter will construct a dVP packet with the packet body containing the data it has been asked to transmit. The dVP packet headers are set and the socket previously created is used to transmit the complete dVP packet. The dVP packet creation and transmission process is shown in the code listing below:

Listing 5.1: Transmission of a captured packet within the dotNetVis Protocol Library

```
public static void Transmit(int dataType, string version, long id,
                            string src, string dest, ushort port, DateTime date)
{
    String pkt = (dataType +
                  ";" +
                  version +
                  ";" +
                  id.ToString() +
                  ";" +
                  src +
                  ";" +
                  dest +
                  ";" +
                  port.ToString() +
                  ";" +
                  date.ToString());
    pktSize = asen.GetByteCount(pkt);
    s.Send(pktMarker);                            //indicate valid dVP packet start
    byte[] meth = { (byte)Method_Types.SEND_PACKET };
    s.Send(meth);
    s.Send(BitConverter.GetBytes(pktSize));       //num of bytes used for pkt body
    s.Send(asen.GetBytes(pkt));                   //send pkt
    s.Send(pktMarker);                            //indaicate end of pkt
    Console.WriteLine("Number of packets sent: " + counter);
    counter++;
}
```

Once the packet processing thread has finished processiong packets from the capture source, the server uses the transmitter to send a notification to the client instructing the client to disconnect from the server. The server shifts into waiting state until another client connection is requested or the server is stopped.

5.2.2 Client Communication

The mainstream data input source available to the client application is through use of the dVP API. This is because the client application's only purpose is to render a visualisation based on data which is processed at the server. The dVP API defines a receiver class which offers the functionality required by a dotNetVis client application.

The receiver component is responsible for capturing and analyzing dVP packets. The receiver provides storage for packet data as well as a variety of retrieval methods. The client application also makes use of the dVP library's Transmitter class to send requests to the server.

Receiving a data stream

When the client application requires data for a visualisation, a connection to the server is attempted. Once a connection is established, the client will request that the server begins a data capture stream based on the specified configuration. This is done by sending a dVP packet with the method header field containing a value of 0x01. Once identified, the server will broadcast on the specified port and the client's dVP receiver will be used to capture all broadcast dVP packets.

The receiver will then store each packet in a list of custom receiver objects. (Due to list capacity limitations, the maximum number of packets the list can contain is 8,388,607[3]). These packet representations are instances of the dVP library's receiver-defined structure named 'Point3D'. This structure is responsible for representing the 5-tuple contained in the body of a dVP packet. The Point3D structure is shown below:

Listing 5.2: The Point3D struct

```
        public struct Point3D : IComparable<Point3D>
        {
            public long ID;
            public float x;
            public float y;
            public float z;
            public long time;

            public Point3D(int _ID, float _x, float _y, float _z, long _time)
            {
                ID = _ID;
                x = _x;
                y = _y;
                z = _z;
                time = _time;
            }
            ///@override: setting comparison of time values
            public int CompareTo(Point3D other)
            {
                return this.time.CompareTo(other.time);
            }
    }
```

A Point3D item represents a packet in the simplest form possible for rendering purposes. It also overrides the CompareTo() method so that objects of this type can be compared by their time values. The ID field is most important as it can be used to refer to the original packet that was captured by the server component.

[3]Based on the type of values the list maintains, as well as the size of each object in the list. A test was constructed to determine this size limit

dVP Packet Processing

The process of populating the receivers with Point3D objects is handled by a packet worker class which is defined in the dVP library. For every dVP packet containing visualisation data received by the client's receiver, a packet worker thread is spawned. This worker thread is given the 5-tuple dVP packet body and it creates a Point3D object using the parameters received.

Each Point3D item created needs to be added to the receiver's list of points. With multiple packet worker threads executing concurrently, access control is required to manage updating of the shared receiver resource (the list of Point3D objects). A simple locking mechanism was implemented to control the update procedure. A snippet demonstrating the use of the lock construct is shown:

Listing 5.3: Thread locking mechanism for access to shared resource

```
class PacketWorker
{
        static readonly object locker = new object();

        public void ProcessPacket(object paramaters)
        {
            String[] p = (String[])paramaters;
            Transmitter.Point3D point = new Transmitter.Point3D();

            lock (locker)
            {
                Transmitter.receivedPoints.Add(point);
            }
        }
}
```

The packet worker class defines a static read-only object which can be locked using the keyword 'lock' in C#. This locks the static object across all thread instances of the class. The thread which locked the static object will unlock it when it has completed the block of code inside the scope of the lock.

The worker threads are used so that the process of receiving packets does not suffer from the additional processing time taken to retrieve and store packet data. The network communication process is also kept separate from the analyzing and storage process.

Once the server has sent all the data it has been requested to process, a 'processing complete' status message will be sent by the server. The receiver will disconnect from the server when it receives this message. It will then initialize the indexing process on the collected data.

Data Indexing

The list of packets is sorted (based on the pre-defined comparison method specified in the Point3D struct) and the minimum and maximum packet arrival times are attained. These are used, along with the minimum replay rate allowed, to set up a list of indices that are used to retrieve subsets of the data based on minimum replay rate time groupings[4]. Due to the volume of data, it is unfeasible to perform time comparisons during visualisation replay.

The methods 'getPoint3DRangeFromNavigationIndex()' and 'getVectorRangeFromNavigationIndex()' are used by the dVManager to retrieve the data subsets for rendering. The vector range method is preferred as it provides the simplest data type possible to represent each point in the 3-Dimensional plane.

5.3 Chapter Summary

The implementation of the dVP library offers a dotNetVis specific framework for communication of data between teh client and server applications.

[4]Minimum replay rate is 1 millisecond per second, so indeces are created based on this necessity. A higher replay rate will combine indeces to allow retrieval based on larger groupings

Chapter 6

Client Application Development

The client application is the focus point of the dotNetVis system. Any meaning that can be inferred from the abstract data processed at the server, will be dependant on the output visualisation at the client. To handle the various complexities, in terms of expected functionality, the client application is divided into four main components. The rendering component and the dVP API are used to receive and visualise data from the server. The settings window provides a simple interface which allows the user to configure the client application and interact with the server component. The fourth component, providing a means of integration, is the dotNetVis manager which enables cross thread communication and overall management of the client application.

6.1 Client Component Manager

The manager class controls all processing on the client side. It is responsible for instantiating and maintaining objects as well as for invoking static behavior across all the sub components of the client application.

On startup, the settings control and the rendering component are intitialised in separate threads due to limitations detailed in Section 6.2. The manager object will use the threads to provide an internal communications infrastructure, allowing data to pass between threads.

6.1.1 Object Communication

As requests are processed, cross thread communication must occur. The manager class passes objects by reference[1] to the threads it owns, thus allowing for simple communication between objects. Static variables are used as flags to communicate object status and to update object configurations.

When a thread needs to interact with a user interface (UI) control belonging to another thread, a less trivial approach is required. Marshalling must be implemented to allow the calling thread's code to execute on the UI thread in the Window's Forms environment. There are two main alternatives for marshalling thread execution; namely synchronous and asynchronous approaches. The only advantage offered by asynchronous marshalling techniques is to prevent a blocking state in which the calling thread waits for the UI thread to complete execution. The level of control over the dotNetVis system at all times during execution removes the opportunity for a blocking state to occur, thereby negating the necessity of asynchronous methods.

The synchronous marshalling is implemented using a delegate method which handles the passing of parameters via an object array. A check is done to determine if the control must be invoked and, once invoked, processing will continue as normal. The listing below demonstrates this procedure:

Listing 6.1: Marshalling to allow cross thread communication

```
public void SetNavigation(bool b, int range)
{
    if (this.trackNavigation.InvokeRequired)
    {
        //Make use of a delegate method to invoke explicitly
        SetNavigationCallback d = new SetNavigationCallback(SetNavigation);
        this.Invoke(d, new object[] { b, range });
    }
    else
    {
        this.trackNavigation.Enabled = b;
        this.trackNavigation.Maximum = range - 1;
        this.trackNavigation.Minimum = 0;
    }
}
```

[1]Using C#'s 'ref' keyword

Apart from internal communication, it is also the responsibility of the manager class to move subsets of data from the dVP receiver onto the graphics device for rendering. The subset ranges are calculated from the replay position specified on the settings component's replay navigation control. This value is changed by the user of the system and on every change, the manager's packet updater thread will execute.

6.1.2 Packet Updater

The packet updater class is the most important component of the client application as it forms the link through which data visualisation can occur. The bottleneck of the entire system exists at this very point since large subsets of data need to be moved at high speeds onto a single graphics device. To alleviate the congestion, a substantial amount of pre-processing is done before the user can start a visualisation replay.

Due to the cycle speed of the rendering component, access to the list of points on the graphics device is kept to a minimum. The updater thread will acquire a subset of data using the specified navigation value. It will convert the full subset of data into the required format before allocating the list to the referenced device's shared resource.

The subset of data acquired will be represented as a list of points, each having associated X, Y and Z values. These values are used to construct a new subset of points in the format required by the rendering component. In the case of the XNA implementation in dotNetVis, a colour is assigned to each point based on a specified axis (the destination port is the default choice). This colour is used to add an extra dimension to the visualisation by offering visual separation in addition to spacial separation offered by the axes.

The following listing shows how data are acquired, formatted, and transferred to the graphics device for visualisation:

Listing 6.2: Moving packets onto the graphics device

```
//add packets within range to local list
foreach (Vector3 v in Transmitter.getVectorRangeFromNavigationIndex(navValue))
{
    if (v.Z != -1)                              //UDP or TCP packet
    {
        vpc = new VertexPositionColor(new Vector3(v.X, v.Y, v.Z),
                                            xc.getColour(v.Y));
        pktsLocal.Add(vpc);
    }
    else                                        //special case: ICMP plot
    {
        vpc = new VertexPositionColor(new Vector3(v.X, v.Y, 0),
                                                  Color.Gray);
        pktsLocalICMP.Add(vpc);
    }
}
//Move the packets onto the device using the referenced XNA component, xc:
xc.packetsToDraw = pktsLocal;
xc.packetsICMPToDraw = pktsLocalICMP;
```

In the case of ICMP traffic, pre-processing of the data will assign a value of -1 as the Z value of an ICMP packet. In this case, the colour grey is assigned to the point and the point is added to a list of other ICMP traffic on the rendering component.

6.2 Rendering Component

The InetVis implementation made use of openGL to provide a three dimensional, immersive scatter plot visualisation to display network traffic. XNA is used to provide the same functionality for the dotNetVis application. While the .NET framework offers native support for three dimensional rendering in XNA, the integration of an XNA component within a Windows Forms environment presents a few issues.

XNA is a framework which was designed for game development. Applications designed in this way follow a specific flow of execution. Once initialized, the application enters a 'game loop' in which all transformations and vertex calculations are recalculated and the resulting output is redrawn. User input for such an application is processed through polling, where the application will check for input at each iteration of the game loop.

Contrary to this, windows forms applications process input based on event-driven procedures where events are triggered when a predefined action occurs. The XNA component has been created within the context of a windows forms application on a separate thread. The client manager handles this initialization as well as communication between windows forms components and the XNA thread.

6.2.1 Plotting Scheme

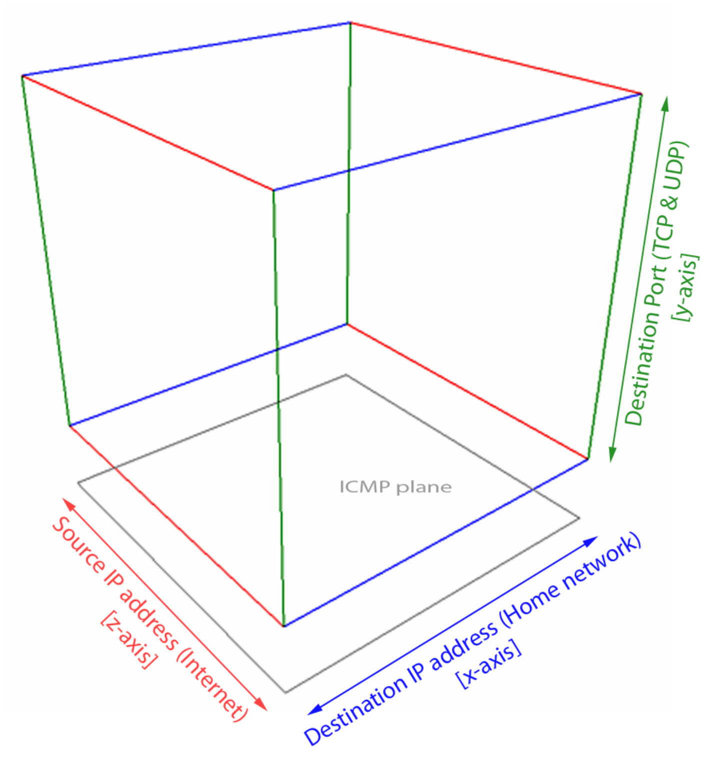

Destination Port (TCP & UDP)
[y-axis]
ICMP plane
Source IP address (Internet)
[z-axis]
Destination IP address (Home network)
[x-axis]

Figure 6.1: dotNetVis plotting scheme

The plotting scheme of dotNetVis is shown in Figure 10. van riel identified this scheme to be an efficient approach at representing network events, associating danger-inferred red with Internet addresses and the calmer blue with the internal network.

Points are positioned and rendered within the dotNetVis cube based on three dimensions. These dimensions are represented as axes and are defined as follows:

- Blue x-axis:

  Packets are plotted along this range according to their destination IP address. Before reading an input stream of network traffic, an IP address range representing the destination subnet must be specified. A ratio is calculated using the packets position within the range. This ratio is then applied to the length of the axis to find a relative position.

- Green y-axis:

  Packets that specify a destination port number (TCP and UDP packets) will be plotted along this range according to that value. Again, the port range must be specified before data is processed. In the case of ICMP packets that have no associated port number, the packet will be plotted on the ICMP plane below the cube.

- Red z-axis:

  Packets are plotted along this range according to their source IP address. Typically, this range represents the full IPv4 Address space, i.e., from 0.0.0.0 to 255.255.255.255, but the user can specify a subset range for filtered monitoring. Again, a ratio is calculated and used to position packets along this axis.

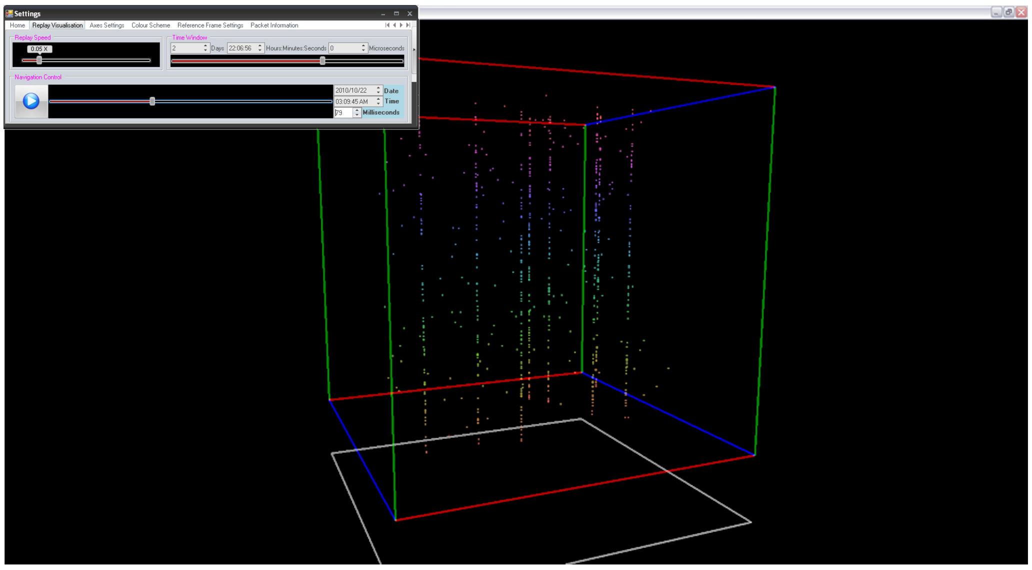
In the case of ICMP packets, a two dimensional plane is sufficient for representation of traffic. To avoid ambiguity within the cube, ICMP traffic is plotted on a separate plane below the cube. This avoids obfuscation and misinterpretation of traffic arriving on ports in the 0-100 range (assuming ICMP traffic would be plotted within the cube at a port value of 0).

With this representation, visual signatures of malicious activity are identifiable. A port scan, as shown in a figure X (a dummy capture, enhanced using photoshop to distinguish points), will take the form of a vertical line; a host scan on a specific port will take the form of a horizontal line spanning the x-axis.

6.2.2 Navigation

Due to the dynamic range and quantity of data that are plotted, a static view of the cube is not feasible. Instead, visualisation navigation is provided to allow the user to dynamically explore subsets of the cube during replay of data.

The InetVis application used rendering techniques such as scaling, transformation and rotation to provide an immersive visualisation. In 3-D programming terms, the world



Replay Speed: 0.05 X

Time Window: 2 Days 22:06:56 Hours:Minutes:Seconds 0 Microseconds

2010/10/22 Date
03:09:45 AM Time
79 Milliseconds

Figure 6.2: A generated visualisation showing visual port scan signatures

(the space in which objects, such as the cube, exist) was changed via navigation. When the user zooms in or out, InetVis will move the cube in the world and subsequently all plotted points would need to be moved as well. In the case of dotNetVis navigation, a camera component is used to view the world. This approach is far more efficient since all navigation is achieved through adjusting the camera object's view of the world, rather than adjusting the world itself.

All navigation of the 3-D space is possible through use of the mouse. This eliminates the need to have visible navigation controls. Boundaries are implemented to limit the user's navigation so that the cube will always be visible on the screen.

6.3 Settings Component

The settings component provides an interface through which the user can interact with and customize the dotNetVis application. The available functionality mimics and extends that of the InetVis implementation while at the same time offering a much improved design in terms of screen usage. The drawback of using the original InetVis application is that the control windows would take up a substantial amount of desktop real estate (monitor space). To demonstrate, on a monitor with a display resolution of 1280x1024, the maximum cube resolution, allowing un-obscured access to controls, is 745x669 (See Figure 6.3). This represents 38% usage of the screen's resolution. With the dotNetVis settings control, the maximum screen usage for representation under the same conditions (excluding the fullscreen option) is 48%.

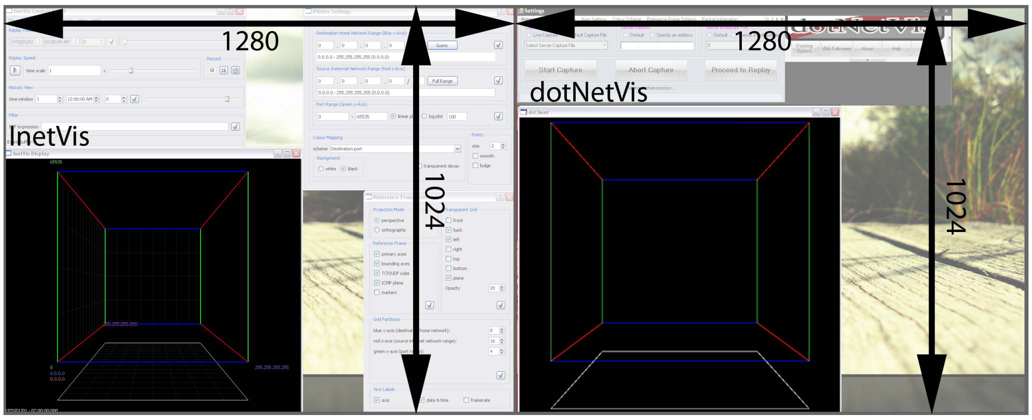
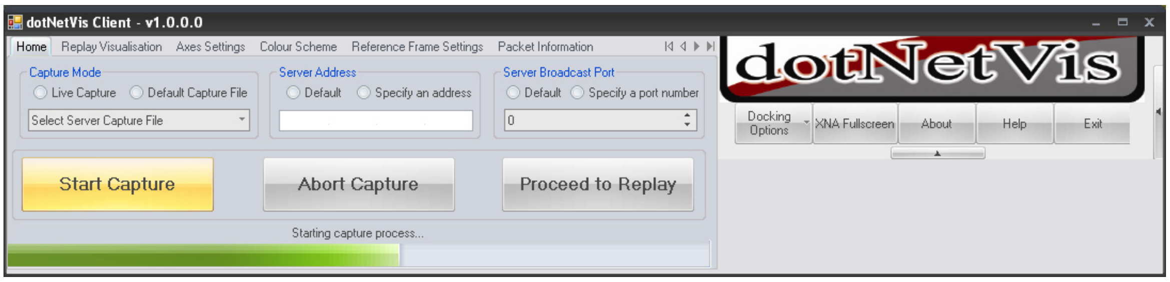


Figure 6.3: Comparison of screen utilization

The redesigned control makes use of tab-based navigation to combine the various windows seen in InetVis, into one simple component. An illustration of each tab's layout, along with an explanation of usage and assignment of values is given below:

- Home Tab

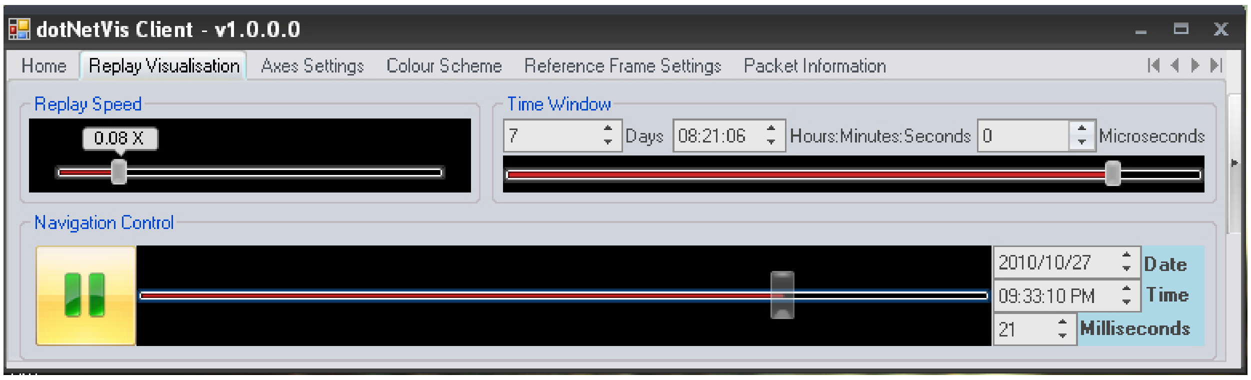


Starting capture process...

Figure 6.4: Home tab layout

The home tab allows the user to configure the server connection settings, to select the network stream source for visualisation, and also to initiate the client server capture process. Once the process has been started, the underlying dVP library will be used to connect to the server using the specified settings and a capture request will be issued. When the capture process is started, a progress bar will relay the state of transmission.

- Replay Visualisation Tab



Replay Speed: 0.08 X

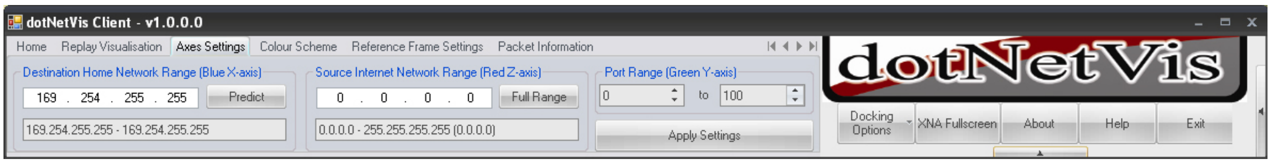
Time Window: 7 Days 08:21:06 Hours:Minutes:Seconds 0 Microseconds

2010/10/27 Date
09:33:10 PM Time
21 Milliseconds

Figure 6.5: Replay navigation tab layout

Replay of capture data will be controlled from this tab. The user can change the replay rate, set the time window indicating the range of data to be viewed simultaneously, and navigate through network events.

- Axes Settings Tab



Destination Home Network Range (Blue X-axis): 169 . 254 . 255 . 255
169.254.255.255 - 169.254.255.255

Source Internet Network Range (Red Z-axis): 0 . 0 . 0 . 0
0.0.0.0 - 255.255.255.255 (0.0.0.0)

Port Range (Green Y-axis): 0 to 100

Figure 6.6: Axes configuration tab layout

The configuration of axes settings is done through this tab. Set the destination range of the home network[2] by either manually entering the range or by selecting the 'predict' button which will attempt to extrapolate the range from the capture file[3]. The current implementation has no validity test to ensure that the correct range has been entered, so care should be taken when manually specifying the range. An incorrect range will simply yield an incorrect visualisation, either displaying no data or a subset of the full set. The destination address of a packet that is to be plotted will determine its position on the X-axis. Set the source internet network range in the same way as the destination range. This can be used to visualize a subset of the traffic in a capture file. The default value will monitor the entire address space. The source address of a packet that is to be plotted will determine its position relative to the Z-axis. This is used to specify a destination port range for monitoring. The destination port number of the IP network traffic that falls within the source and destination network ranges will be plotted on the Y-axis.

- Colour Scheme Tab

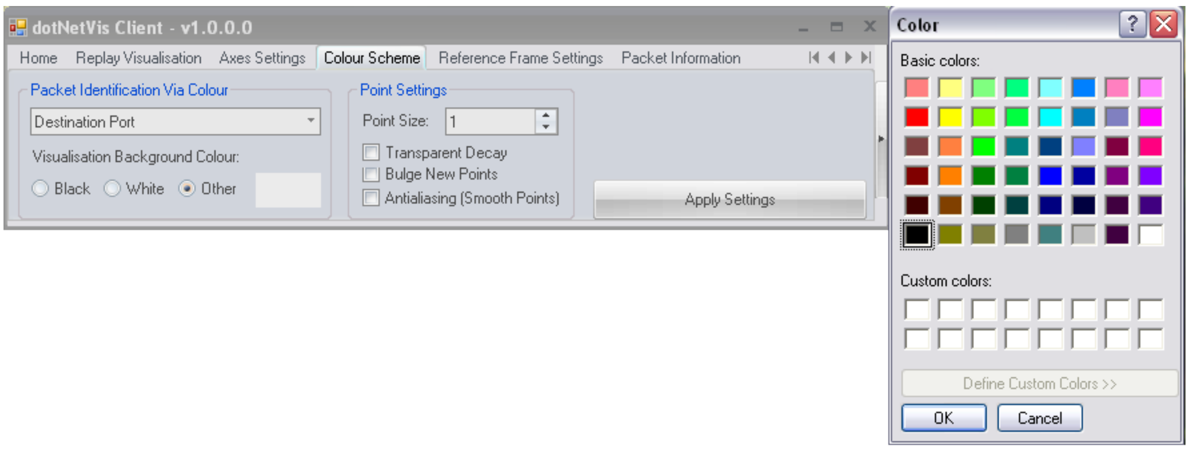


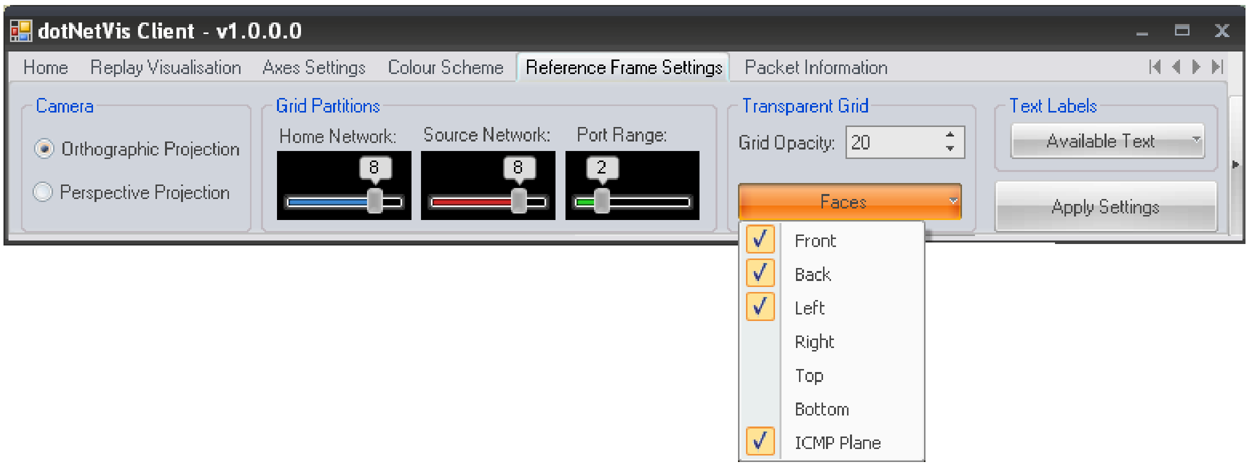
Figure 6.7: Colour scheme choice tab layout

[2]The home network range in this case refers to the address range that was used by the packet capturing device to monitor and record network traffic. The destination address of every packet within the capture file will fall within this range. The range is completely independent of the machine on which the dotNetVis client application is run.

[3]This is not recommended as the approximated range will nearly always be a subset of the full destination range. This is due to the upper and lower destination addresses not being the targets of network traffic, thus not being included in the resulting capture file

The default scheme is for the colours of plotted points to be chosen based on the destination port number value of the associated packet. Other options include using the protocol type, the destination or source address of packets to distinguish between the points. The background colour of the cube can be selected using the dropdown box or by selecting a common white or black setting. The visual representation can be enhanced by selecting an appropriate background colour based on the chosen colour settings and cube congestion. Point settings are included in the design but configuration has not been implemented due to a lack of support in XNA and time constraints.

- Reference Frame Settings Tab



[● Orthographic Projection] ○ Perspective Projection

Home Network: 8
Source Network: 8
Port Range: 2

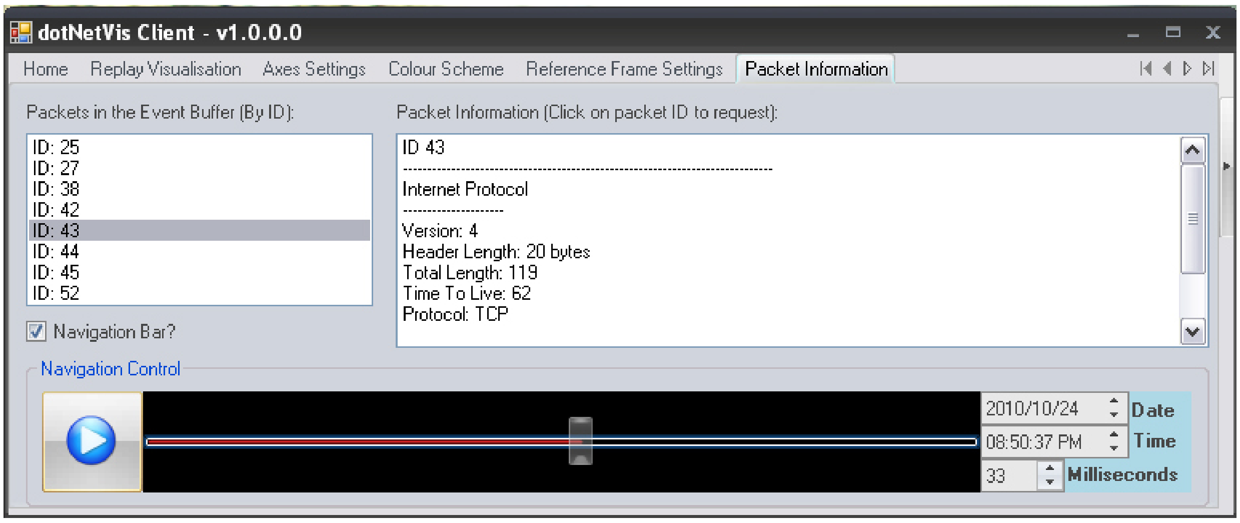
Grid Opacity: 20

✓	Front
✓	Back
✓	Left
	Right
	Top
	Bottom
✓	ICMP Plane

Figure 6.8: Reference frame configuration tab layout

The six sides of the cube, as well as the ICMP plane have the option of displaying a transparent grid that can be used to view the cube in segments rather than as a whole. A drop down checklist is used to toggle these grids. The opacity of the grids can be set to a value between 0 (being invisible) and 100. Each axis can be assigned a value which specifies the number of partitions the user wants the grid to be split into on that axis. This is done using the colour coded sliders. This helps to partition the cube nicely. The default value is 10 partitions on each axis. Text labels can be switched on for a more detailed representation. Axis labels show the ranges that have been selected and they also show the colours that have been assigned to the axes. A date and time label shows the current position of playback in the capture file. The framerate label gives an indication of the performance of the rendering component.

- Packet Information Tab



ID: 25
ID: 27
ID: 38
ID: 42
ID: 43
ID: 44
ID: 45
ID: 52

ID 43

Internet Protocol

Version: 4
Header Length: 20 bytes
Total Length: 119
Time To Live: 62
Protocol: TCP

2010/10/24
08:50:37 PM
33

Figure 6.9: Packet Information tab layout

This tab is useful as it lists all the packets that are currently on display and that are in the event buffer. This list is updated on every frame. By scrolling over the items in the list of current packets, the user will be able to identify the relative packet in the cube. By selecting a packet in the list, that packet's information will be retrieved from the server and will be displayed in a second list. A pause and play toggle button is also available on this tab to allow for playback and navigation while concurrently observing the contents of the packet event buffer.

6.4 Chapter Summary

Having described all components, the overall flow of data has been described from input at the server components through to the output visualisation at the client's rendering component.

Chapter 7

Conclusion

7.1 Overview

In conclusion, we have re-implemented a relatively full featured and highly optimised port of the InetVis tool. The dotNetVis tool leverages the XNA framework for the cube generation while utilizing the .NET framework's GUI controls to capture input and display overhead information on a per packet scale. The processing of packets is handled completely separately to the dotNetVis server. The dotNetVis protocol allows for uninterrupted communication between client and server.

We begin this section with an overview of the current feature set of dotNetVis. Following, is a section which outlines future, desirable functionality currently not implemented within this system, either due to complexity or time constraints. A section mentioning the performance results of the dotNetVis system, highlighting the improvement on the InetVis tool, is given and in closing, a summary of this both this document and the success of this project is given.

7.2 Current Functionality

The dotNetVis server allows for the processing of IP packets which are captured by a network telescope. The processing that the packets undergo includes filtering and extracting necessary information that needs to be passed onto the dotNetVis client.

Packet data is sent using the dotNetVis protocol and once received by the client, an XNA representation of the packets is shown. If the packets are being read from a capture file, a variable playback rate is used. This can be set in both the client and the server applications. Playback rates range from 0.001 times (one millisecond per second) to 86400 times (one day per second) just as the InetVis tool allowed.

Navigation of the 3D environment is also enabled through rotation and scale transformations applied to a camera in the XNA world. This allows users to view the 3D cube in a 2D perspective. Once the user identifies a packet of interest, they can use the mouse to select the packet from a list of plotted packets displayed on the settings component. Once a packet is selected, the packet ID is used to retrieve information on the selected packet. This information is displayed in the information tab of the custom control.

In the same way that InetVis allows, dotNetVis offers a customizable view of the cube. A user can set the background colour, the axes colours and text colours to enhance playback. Reference grids and axes can also be turned on or off. An orthographic or a perspective camera view can be used to view the cube for different purposes. The user controls the ranges that the cube draws so if a packet is received from the server that falls out of range, it is not drawn.

7.3 Future Functionality

- Variable Logarithmic scaling

  The current implementation of dotNetVis does not support the logarithmic axes scaling methods used in InetVis. With a large portion of traffic falling into the port range of between 0 and 1023, the lower parts of the cube are not easy to read. Using logarithmic scaling to expand the lower port numbers, the cube can be much more easily interpreted.

- Web Client

With the client server approach implemented in dotNetVis, a very useful extension would be a web based client for the dotNetVis tool. With the advancement of web development and the continual growth of bandwidth, a web based 3D rendering client is not far off. Working with the dotNetVis protocol, a web client can easily access the server data and render a cube. Using a XAML browser application approach, 3D graphics can be displayed using xml files. By configuring the dotNetVis server to output xml, this approach can be used to produce a web client visualisation.

- Cube signature detection and representation

  An extension of this magnitude would transform the dotNetVis tool into an Intrusion detection system (IDS) as the client would then have the ability to identify malicious activity. As seen in the figure x, malicious signatures are easy to spot with the human eye. This extension would involve providing an intelligent component which would monitor the plotted packets and infer malicious signatures from the data the cube is displaying.

- 3D Navigation

  - Section expansion capabilities
    Navigation and packet selection is not optimal in the current implementation of dotNetVis. By allowing a user to expand a portion of the cube, a much more accurate navigation experience can be seen.
  - 3D hardware support
    The XNA environment allows for simple integration between the XNA component and various hardware devices. With the boom of 3D, new devices are being released, and there are already a few available, which allow for 3D navigation, such as 3D mouses. By using such a device, navigation could be less tedious and more natural.

- Updated client side front-end

  As was mentioned with the web client extension, building a client application is relatively simple as none of the packet capturing and issuing needs to be re-implemented. A client can use the dotNetVis protocol to communicate with the server and once it receives data from the server, it merely has to render the data in some meaningful manner. This flexibility allows for a multitude of different graphs which can be tailored for specific needs.

- Filtering with BPF expressions

  InetVis made use of BPF filtering to filter traffic for replay. While dotNetVis has been implemented inbuilt filtering, there exists no option to apply filtering on the client side. This could be useful for when a client wishes to view specific traffic within a replay visualisation.

- Avi support

  It would be a useful option to allow a user to record playback of a visualisation. This will allow later replay of the visualisation without having to invoke the capture process at the server side again.

7.4 Performance Results

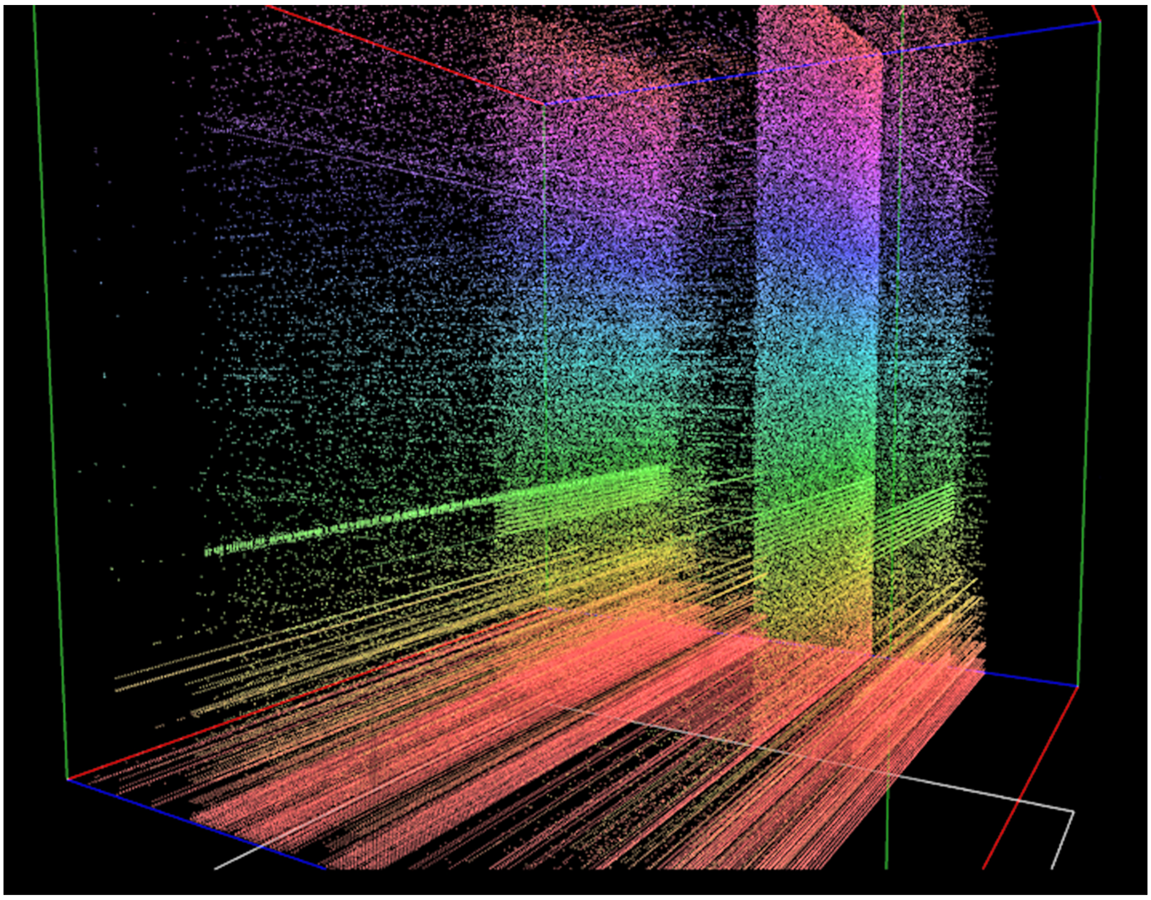


Figure 7.1: 8,200,000 packets displayed

It is shown in Figure 7.1 that the representation of large datasets is no longer an issue in the dotNetVis system. The server component captured and transferred 8.2 million packets in just over 5 minutes from the dump file containing data representing the traffic on Rhodes University's class C network telescope over 2009. This large dataset (a magnitude of 10 times the maximum size the InetVis tool could manage) was displayed at a decent 7 frames per second by the client application. The client process consumed 728mb of RAM to allow for storage of these packets, and CPU usage idled around 60% across all four cores. The machine on which the client was run has 3.2 GB of effective RAM (due to the 32bit environment of the operating system) and a 2.66GHz Intel Q9400 quad core processor.

As was mentioned in Section 2.4.2, the InetVis tool was tested on a machine with 1 GB of RAM and a 3.0GHz processor and only managed to display some 450,000 packets before system performance became unstable.

7.5 Summary

While no improvements on the underlying concept were implemented, a host of performance enhancements were carried out. Optimization of processor usage, modularization of the input and output process and improvements on memory usage coupled with increased memory capacity are some of the enhancements that improve on JP van Rielś original 3-D scatter plot visualisation tool - InetVis. This research has proved successful in optimizing the InetVis system to handle much larger datasets as well as offer a much more flexible architecture on which future extensions can be built.

Bibliography

[1] Cisco visual networking index: Forecast and methodology, 2009 to 2014. Tech. rep., Cisco Systems, Inc, 2010.

[2] Bro nids. Online article, Online article : *http://www.bro-ids.org* [Accessed : 18/07/2010].

[3] Internet usage statistiacs: The internet big picture, October Online article : *http://www.internetworldstats.com/stats.htm* [Accessed : 29/10/2010].

[4] ”tcpdump - dump traffic on a network man page., Online article : *http://www.tcpdump.org/* [Accessed : 16/10/2010].

[5] Tcpdump packet capture utility and libpcap packet capture library. Online article, Online article : *http://www.tcpdump.org* [Accessed : 20/07/2010].

[6] Wireshark - the world’s foremost network analyzer, Online article : *http://www.wireshark.org/* [Accessed : 16/10/2010].

[7] Bailey, M., Cooke, E., Jahanian, F., Myrick, A., and Sinha, S. Practical darknet measurement. 1, 6.

[8] Becker, R. A., Eick, S. G., and Wilks, A. R. Visualizing network data. *IEEE Transactions on Visualizationand Computer Graphics 1* (1995).

[9] Conti, G. *Security Data Visualisation.* William Pollock, 2007.

[10] FISK, M., SMITH, S., WEBER, P., KOTHAPALLY, S., and CAUDELL, T. Immersive network monitoring. *PAM2003 Passive and Active Measurement 2003, NLANR/MNA (National Laboratory for Applied Network Research / Measurement and Network Analysis Group)* (2003).

[11] Fossi, M. Symantec global internet security threat report - trends for 2009. 48.

[12] GAL, T. Sharppcap, Online article : *http://www.tamirgal.com/blog/page/SharpPcap.aspx* [Accessed : 03/04/2010].

[13] HARDER, U., JOHNSON, M. W., BRADLEY, J. T., AND KNOTTENBELT, W. J. Observing internet worm and virus attacks with a small network telescope. 1,2,14.

[14] KUROSE, J. F., AND W.ROSS, K. *Computer Networking - A top down approach.* Greg Tobin, 2008.

[15] LAU, S. The spinning cube of potential doom, Online article : *http://www.nersc.gov/nusers/security/TheSpinningCube.php* [Accessed : 22/07/2010].

[16] MARTY, R. *Applied Security Visualisation.* Pearson Education, Inc., 2008.

[17] MOORE, D., SHANNON, C., VOELKER, G. M., AND SAVAGE, S. Network telescopes: Technical report. 1–2.

[18] MUELDER, C., MA, K.-L., AND BARTOLETTI, T. Interactive visualization for network and port scan detection. 1 – 4.

[19] STEFAN AXELSSON, D. S. *Understanding Intrusion Detection through Visualisation.* Springer Science+Businss Media, Inc, 2006.

[20] TAMASSIA, R., PALAZZI, B., AND PAPAMANTHOU, C. Graph drawing for security visualization. 1–3.

[21] VAN RIEL, J. Multi-dimensional visualization for the analysis of internet traffic and the identification of intrusive activity. Master's thesis, Rhodes University, 2005.

[22] VAN RIEL, J.-P. Inetvis, a visual tool for network telescope traffic analysis. 1–5.

[23] VAN RIEL, J.-P., AND IRWIN, B. Toward visualised network intrusion detection.

[24] XIAO, L., GERTH, J., AND HANRAHAN, P. Enhancing visual analysis of network traffic using knowledge representation.

[25] YIN, X., YURCIK, W., TREASTER, M., LI, Y., LAKARAJU, K., AND LAKARAJU, K. Visflowconnect: netflow visualizations of link relationships for security situational awareness. *In Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security ACM Press* (2004), 35 – 44.

Appendix A

Library Code Listing

Listing A.1: Library definitions

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Collections;
using System.Threading;
using Microsoft.Xna.Framework;
using System.Diagnostics;

namespace dVP
{
    public class Transmitter
    {
        private static bool generateRandomPackets = true;

        private bool connected = false;
        private bool _shouldStop = false;
        private static bool capFileRequested = false;
        private bool capFileProcessing = false;
        private String ServerIP = "146.231.123.91";
        private int ServerPort = 31337;
        TcpClient tcpclnt;
        Stream stm;
        public static List<Point3D> receivedPoints = new List<Point3D>();
        public static List<Vector3> vectors = new List<Vector3>();
        public static int[] indeces;
        public static int navigationRange = 0;
        public String[] RECEIVE_PACKET_FIELDS = {  "Method:",
                                                   "Version:",
                                                   "Packet ID:",
                                                   "X int:",
                                                            "Y int:",
                                                            "Z int:",
                                                            "Microseconds:"};
        public static float xRange = 4294967296.0f;
        public static float yRange = 4294967296.0f;
        public static float zRange = 65536.0f; //max is 65536
        public bool isReady = true; //when true, a batch of packets has either been
            processed or the packet array is empty
        public static long maxTime = 0L;
        public static long minTime = 0L;
        //public static int navigationRange = 1000;
        private BufferedStream bstm;
        enum StatusReplys : byte
        {
            OK = 0x00,      //the servers request is approved
            WAIT = 0x01     //the server must wait for its default time and try again
        };
        private byte[] pktMarker = { 0xFF };
        public enum Method_Types : byte
        {
            SEND_PACKET = 0x00,
            REQUEST_DEFAULT_CAP_FILE = 0x01
        };
        int COUNT = 0;
        Stopwatch stopWatch = new Stopwatch();

        public struct Point3D : IComparable<Point3D>
        {
            public long ID;
            public float x;
            public float y;
            public float z;
            public long time;

            public Point3D(int _ID, float _x, float _y, float _z, long _time)
            {
                ID = _ID;
                x = _x;
                y = _y;
                z = _z;
                time = _time;
            }

            public int CompareTo(Point3D other)
            {
                return this.time.CompareTo(other.time);
            }
        }

        public void ThreadRun()

        private void waitForCapFileRequest()

        private void connectToServer()

        private void startListening()
```

```
        private void receive()

        private void finishProcessingCapFile()

        public static int getNavigationRange()

        private void disconnectFromServer()

        private void transmit(byte b /*request based on method_types enumerator*/)

        private void transmit(String s)

        public static long ToInt(string addr)

        static string ToAddr(long address)

        public static void requestCapFile()

        public bool isPacketArrayReady()

        public void generateRandoms()

        public static void createIndeces(int range)

        public static void createVectorList()

        public static List<Point3D> getPoint3DRangeFromNavigationIndex(int i)

        public static List<Vector3> getVectorRangeFromNavigationIndex(int i)

        public void RequestStop()
    }

    class PacketWorker
    {
        static readonly object locker = new object();

        public void ProcessPacket(object paramaters)
        {
            String[] p = (String[])paramaters;
            Transmitter.Point3D point = new Transmitter.Point3D();
            point.ID = Convert.ToInt64(p[2]);
            point.x = Transmitter.ToInt(p[3]) / Transmitter.xRange;
            //Console.WriteLine("time: " + Convert.ToInt64(p[5]));
            long temp = 0;
            try
            {
                temp += (Convert.ToInt64(p[4].Substring(0, p[4].IndexOf('.'))) * 255 *
                    255 * 255);
                p[4].Remove(0, p[4].IndexOf('.'));
                temp += (Convert.ToInt64(p[4].Substring(0, p[4].IndexOf('.'))) * 255 *
                    255);
                p[4].Remove(0, p[4].IndexOf('.'));
                temp += (Convert.ToInt64(p[4].Substring(0, p[4].IndexOf('.'))) * 255);
                p[4].Remove(0, p[4].IndexOf('.'));
```

```
                    temp += (Convert.ToInt64(p[4].Substring(0, p[4].IndexOf('.'))));
                }
                catch (Exception e) { Console.out.Write("The IPAddress cannot be pasrsed") }
                point.z = temp / Transmitter.yRange;
                if (p[0].CompareTo("1") == 0) //ICMP packet received
                    point.y = -1;
                else point.y = Convert.ToUInt16(p[5]) / Transmitter.zRange;
                point.time = Convert.ToDateTime((p[6])).Ticks;
                point.x = (point.x * 2) - 1;
                point.y = (point.y * 2) - 1;
                point.z = (point.z * 2) - 1;
                lock (locker)
                {
                    Transmitter.receivedPoints.Add(point);
                    //Console.WriteLine("Packet list size: " + Transmitter.packets.Count);
                }
            }
        }
    }
```