Determining Accurate Network Flow Endpoints

A dissertation

submitted in partial fulfilment

of the requirements for the Degree

of

Postgraduate Diploma

at the

University of Waikato

by

David Murrell

University of Waikato

2012

Abstract

This report undertakes a review of the available approaches to network flow timeouts, and seeks to systematically compare algorithms using previously collected network traces from the WITS archive (WAND, 2011).

This report evaluates the algorithms from two perspectives: A Traffic Accounting Service, and that of a Network Firewall. It also makes observations on how different methods are better suited to different applications.

The overall aim of this project is to establish a set of tools for future research in this area, by extending the Libflowmanager application (Alcock, 2011).

Acknowledgements

I wish to thank my supervisor, Tony McGregor, for his valuable advice and kind words during the many hours of his time we spent discussing this project.

To my partner, Sarah Ward, thank you so much for your steadfast support during this year long project.

Contents

1 Introduction **1**

2 Background **3**

2.1 The Underlying Network 3
2.2 Physical and Link Layer 5
2.3 Internet Layer 5
2.4 Data Transport 6

3 Algorithms **11**

3.1 Cisco Netflow 12
3.2 Fixed Timeout 14
3.3 Fixed Timeout TCP Variation 15
3.4 Internet Standards Based Timeout 15
3.5 MBET 17
3.6 Netramet 19
3.7 Netramet-Short 21
3.8 Dynamic Timeout Strategy (DoTS) 21
3.9 Probability-Guaranteed Adaptive Timeout 21
3.10 Quan and Carpenter (2011) 24
3.11 Discussion 24

4 Comparative Research Process **25**

4.1 Libflowmanager 25
4.2 Algorithm Implementation 26
4.3 Real World Data 29
4.4 Symphony Cluster 30
4.5 Post Processing 32
4.5.1 Broken and Uneven Flows 38
4.5.2 Further Performance Work 39
4.6 Statistics 39
4.6.1 General Trace Statistics 40
4.6.1.1 Memory Time and Total Flows 40

4.6.1.2 Broken and Uneven flows 40
4.6.2 Algorithm Specific Statistics 40
4.6.2.1 Time Wasted 40
4.6.2.2 Shortflows 41
4.6.2.3 Shortened 41
4.6.2.4 Lengthening 41
4.6.2.5 High Water Mark 42
4.7 Discussion 42

5 Results **43**
5.1 Accounting Results 44
5.1.1 Memory Highwater Mark 45
5.1.2 Total Flow Memory Time 48
5.2 Firewall Results 50

6 Future Work **53**

7 Conclusion **55**

Appendices **55**

A MBET Configuration Options **56**

B DoTS timeout Table **57**

C Software **58**

Chapter 1

Introduction

This dissertation examines how an algorithm can determine when network traffic exchanged between two applications has finished.

Any algorithm attempting to determine when network traffic has stopped being exchanged between applications will be constrained by having no direct information about the internal state of either of the two applications transmitting data. Any information about when an application is about to stop transmitting must be contained in or about the packets of data flowing past the algorithm observation point.

End of flow algorithms are used in almost all pieces of network attached hardware, and as yet, no systematic approach to validating their effectiveness has been completed. Additionally, no set of publicly available software tools to compare the effectiveness of existing algorithms to newly developed algorithms are known to exist. This research aims to fill some of this gap by reviewing the published literature and developing the tools required to evaluate the effectiveness of the algorithms identified in the literature review.

This dissertation will evaluate the results of running the algorithms on traces using two scenarios: a Traffic Accounting System and a Network Firewall. These scenarios have different purposes, and therefore have different objectives that the algorithm powering them should meet.

This dissertation evaluates how algorithms respond to changes in traffic

over time, and if they continue to be effective in a changing network environment.

This report includes seven chapters:

- Chapter 1, Introduction, this chapter. This introduces the paper, and explains why this work has been undertaken.
- Chapter 2, Background, describes the general concepts of networks and flows; and explains the constraints around packet observation and how this impacts determining when flows end.
- Chapter 3, Algorithms, details the flow termination algorithms that have been discovered through a literature search, and explains them in detail.
- Chapter 4, Comparative Research Process, explains the implementation and methodology for subsequent comparison of the output of the algorithms.
- Chapter 5, Results, which details the output of the comparison process.
- Chapter 6, Future Work, looks at areas of study that this report may lead into.
- Chapter 7, Conclusion, summarises the overall contribution and impact of this work.

Chapter 2

Background

As mentioned in the introduction, this research evaluates algorithms that attempt to determine when traffic has stopped being exchanged between two applications on a network. Traffic on a network consists of individual packets that each contain layers of information, allowing the transmission of data on a network. When two applications exchange data, the set of traffic that is exchanged between the two applications is named a flow.

A flow is the formal term for a set of related data packets transmitted between two applications, and is specified by a set of five values present in each packet of data, unique to that flow. One flow is considered to be different to another flow by having one or more differences in any of the five identifying values that make up a flow descriptor. The five identifying attributes are also formally known as a 5-tuple.

The timeout algorithms detailed in Chapter 3, Algorithms, use the 5-tuple as flow attributes of data packets, so a further explanation of how these identifiers are important in terms of network traffic is presented below.

2.1 The Underlying Network

This research centres on Internet Protocol (IP) networks, where data transmission occurs in a series of data packets, exchanged between two points (or hosts) on a network. The first host to send a packet to another host is typically

termed the source host, and the host that receives the packet is termed the destination host.

Network packets are split up into independent sections, called layers, that each concern themselves with different objectives. Implementations of these layers are called protocols. The difference between a layer and a protocol is that the layer is an abstract demarcation point in a packet; a protocol documents what fits inside that space. It follows then, that there can be more than one protocol for a layer. Examples of this pattern follow.

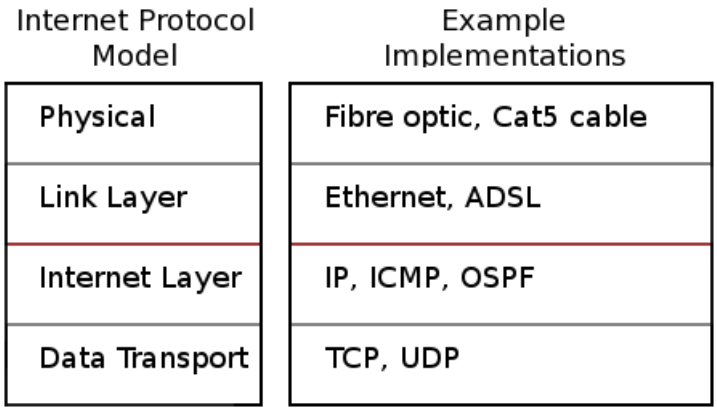

Figure 2.1: A Network Packet Model

Figure 2.1 describes the logical layout of a data packet, and some of its transmission methods. This is a cut down diagram, as the Internet Protocol model listed has other layers that discuss how an application interacts with a packet. For the purposes of an algorithm deciding when to end a flow, the algorithm will only use data from the Internet and Data Transport layers. The Internet Protocol Model does not include specifications for Physical and Link Layer protocols, but examples of what sits before the Layers of an Internet Protocol packet are useful for understanding how data is delivered onto a network.

2.2 Physical and Link Layer

At the physical layer, data is copied from one location to another typically either using a part of the electromagnetic spectrum over optical fibre or air, or electrical impulses over copper wire. The Link layer specifies how a physical transmission interface copies packet data between two points, ensuring that what is sent at one side of a link is successfully received at the other side of a link. Ethernet over copper Cat5 networking cable and ADSL over a pair of wires terminated at a residential dwelling are examples of this.

2.3 Internet Layer

The next section of a network packet is the Internet Layer. Its primary role is to contain the addresses and other data to enable network routing devices to deliver a packet from one destination to another. An Internet Network is a series of devices connected to each other using non-homogeneous physical and link layer protocols, but which communicate with each other using a standardised Internet Layer Protocol. An iPad gaining access to the Internet via an ADSL broadband router using a WiFi is a common example of the above. An Internet Layer packet from the iPad will traverse an air gap to the ADSL modem on the wall, copper wiring to the roadside cabinet, and fibre optic cable used by the household telecommunications provider before being routed onto the Internet.

The Internet Protocol (IP) is an implementation of the Internet Layer. A device on an IP network has its own distinct IP address, meaning that any device on a IP network can address any other device and exchange routed packets of data with that specific device. This is an efficient approach to network utilisation that allows many more devices to exchange data than they would be able to than if network communication was via a broadcast system. An IP packet header contains both source and destination IP addresses, allowing devices on an IP network to trivially find the original sender of a IP

packet and send IP packets back to the source of a received packet. IP routing devices (routers) are highly connected devices that sit on an IP network, and pass traffic from one interface to another with the intention of moving a processed packet away from a packet's source address and onwards towards a packet's destination address. Routers achieve this by reading the IP header of a packet, and forwarding the packet out an interface that will put the packet closer to its eventual destination on an IP network.

Hosts are devices with a single IP address, typically connected at the edge of an IP network, and typically only have one connecting interface with which to exchange traffic with a connected IP routing device. The Internet Protocol network does not guarantee packet delivery, but instead works on a best effort basis, where packets are delivered if it is possible to do so, but may be deleted without notice by routers or hosts if congestion or link problems are encountered. Routers will also delete packets to keep links from becoming excessively congested, and to ensure the stability of the network as a whole. This deletion of packets is termed packet loss, and is a common occurrence on IP networks.

Another protocol that can occupy the Internet Layer is called the Internet Control Message Protocol (ICMP). It is used to transmit control messages between hosts or routers on an IP network. ICMP can be used to contain reference information for end hosts to help in diagnosing problems with connectivity on a IP network. An algorithm looking to timeout a flow will use the source and destination IP address of a packet as two of its five identifiers.

2.4 Data Transport

The Data Transport Layer is located further into a packet. The role of the Data Transport Layer is to provide a data transport function for a network packet so that applications running on hosts connected to an IP network may exchange data with each other. A host usually has more than one application running on it, so an application is allocated an identifier, called a port, which is used

to distinguish it from other applications on a single host. A Transport Layer protocol will have two sections for port information, so that the destination application has an address on a host with which to address data to be sent back to the source application.

The two principal data Transport Layer protocols in use on the Internet are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). UDP specifies the a source and destination application port on a host that the data is bound for and a place to put payload data for an application. Any further checking or data retransmission is handled by an application after it receives the data. TCP was developed to enable reliable transport of data between two applications, so that an application can specify an application on a remote host to send data to, and be notified that the data has successfully reached its destination.

A successfully delivered TCP packet will have a corresponding TCP packet sent in the opposite direction containing an acknowledgement (ACK) flag. When the source host receives an ACK flagged packet, the source host can guarantee that the previously sent data has been delivered correctly.

TCP also includes start and end of flow information. The beginning of a TCP flow is marked with synchronize (SYN) flags exchanged in both directions, followed by corresponding ACK flagged packets. This series of SYN, SYN/ACK and a final ACK packets is called the three-way TCP handshake. At this point, both hosts will consider the connection to be established, and will exchange application data. To close an established connection, hosts will send each other finalise (FIN) flagged packets, followed by ACK flagged packets, after which the connection will be considered closed by both hosts. UDP does not define a reliable transport mechanism like TCP does, and as such, applications that use UDP for communications must implement their own methods of ensuring successful data delivery.

A timeout algorithm will use the implemented protocol type as a distinguishing value, typically set to either TCP or UDP. Additionally, the other

values used are the source and destination ports inside a Transport Layer protocol. These five values (IP Source and Destination Address, Source and Destination Port, and Transport Protocol) are used by timeout algorithms to team related packets together into a flow of traffic. As noted at the beginning of this chapter, this combination of identifiers is termed a 5-tuple.

When evaluating traffic flowing between two end points, a decision has to be made about whether to evaluate traffic flowing in one direction independently to the other direction (unidirectional), or to evaluate both directions as related traffic (bidirectional). This dissertation chooses to concern itself with bidirectional flows, since the application context that the algorithms will be evaluated in is generally bidirectional.

Determining the start and end of a flow for a TCP flow is nominally trivial. All TCP flows will start with a 3-way TCP handshake denoting the beginning of a flow and the first packet will contain a unique 5-tuple, differentiating it from other flows. This is very rarely the observed case. Most network links in an operational environment as well as available traces for research purposes will have had traffic flowing prior to packets being captured, meaning that the three way TCP handshake will not be seen for established TCP flows. Even if it were possible to ensure that all TCP flows on a link were newly established after analysis of the traffic on the link had started, UDP has no such start of flow information, and the first UDP packet seen containing a unique 5-tuple has to be considered the beginning of a flow. For the purposes of this research when evaluating algorithms, this dissertation considers the first packet with a particular 5-tuple value, be it UDP or TCP, signifies a new flow, regardless of any flags set in the packet. Complicating matters further is packet loss, as referenced in section 2.3, Internet Layer. This is an ongoing problem for a flow timeout algorithm, because a packet that is marked as being part of a flow may not reach its destination, causing an algorithm to mark a flow as being in a different state to what the endpoints think the state of the flow is.

Additionally, accurately determining the actual endpoint of a flow in the

face of a best-effort network that can arbitrarily drop packets at any point en route to their destinations is therefore a non-trivial problem. UDP contains no end of session information, and any observed packet for a flow has to be considered to be the possible end of that flow. With TCP, this problem can be somewhat alleviated by noting that if a TCP flow is established and exchanging packets, a set of FIN flagged packets should occur in the flow at some point in the future, denoting the endpoint of the TCP flow between the two hosts. One of several issues with this assumption is that the connected host may not get a chance to successfully close all its TCP sessions before being permanently disconnected from a network, in which case it will never send the requisite FIN flagged packets, and the flow will never be marked as closed. This underlying limitation means that the various TCP session tear down flags can be used for hinting the end of a flow, but must be considered in terms of observed packets, rather than any assumption that the end of flow markers will always arrive.

Other issues affecting observed network traffic are packet capture device only capturing one direction of a flow due to asymmetric routes; host firewalls timing out established connections and dropping or rejecting later incoming packets; and other issues like upstream route changes that deliver ranges of IP addresses to their destinations via an alternative network path not visible to the observer, causing a series of seemingly established flows to suddenly go quiet.

These network issues can cause packets to either be invisible to the observer, or can imply to an observer a different state on the end hosts than actually exists.

For these reasons, a flow is normally defined as active until a terminating condition is reached. A simple terminating condition is that no packets related to flow have been seen for some fixed time, and the flow is marked as timed out. Many other flow termination algorithms have been proposed in the literature including MBET (Ryu et al., 2001), DOTS (Ming-zhong et al., 2006) and Netramet (Brownlee and Murray, 2001).

The goal of this project is to develop a framework that allows the performance of different flow termination algorithms to be compared, taking account of the different potential uses of a flow and different network traffic profiles that may occur.

The next chapter looks at differing approaches to how parts of packet data can be used to estimate an end of flow problem, how they originated, and, with some detail, how they work.

Chapter 3

Algorithms

The literature search identified nine distinct approaches to end of flow estimation. Two of these approaches have a variation that significantly modifies their behaviour, so the variations have been investigated as well. All flow timeout algorithms follow the same basic format:

1. Check if packet is part of a flow or create a new flow entry
2. Add the packet information to a flow entry
3. Recalculate the timeout value for a flow
4. Update the timeout for the flow
5. Check the flow to see if it is to be expired
6. Check the other flows to see if they need to be expired

Step 4 is always applied when a flow is updated with new packet details. This resets the timeout value on a flow to a new time in the future. This new reset time is calculated from when the packet was received, plus the timeout value that the algorithm has specified for the flow. In the case of the Fixed Timeout Algorithm in subsection 3.2, this resets the timeout for the flow to be a static value (typically 64 seconds) in the future.

The recalculation of what the timeout value should be, as mentioned in step 3, is another implementation-specific detail. Some algorithms delay the

recalculation to particular named steps for a packet. An example of this is Probability-Guaranteed Adaptive Timeout Algorithm, detailed in subsection 3.9. PGAT only recalculates timeout based on a predefined interval of the first packet, 10th, 100th, and 500th packet.

Step 6 is an algorithm specific detail: Some algorithms delay the check of all other active flows when one flow is updated to save on CPU resources. This has two direct drawbacks: firstly, the cost of extra memory usage when extra flows are added and none are removed from memory, and secondly, leaving flows in memory that are past their expiry time. This trade off can be acceptable for an accounting system, since memory can be cheaper than CPU time. However, the impact of leaving a flow open too long on a firewall can be a security risk, as an open flow record could allow attackers to send unexpected packets through to a host that is not expecting them.

Figure 3.1 shows the dependency graph of the algorithms found during the literature review, showing how flow endpoint analysis concepts have developed over time. Netflow is not linked to any of the other timeout algorithms as it was developed independently of the other algorithms, though it shares some common aspects of the Fixed Timeout scheme. What is interesting about this graph is that the DOTS (Ming-zhong et al., 2006) and PGAT (Wang et al., 2005) algorithms, while being developed later than the Netramet algorithm (Brownlee and Murray, 2001), do not utilise the ideas presented in the Netramet paper, and only seek to improve on the algorithm presented in the MBET paper.

3.1 Cisco Netflow

Netflow was originally developed by Cisco as a method of caching flow routing decisions (Cisco, 2007). This flow information cache was later extended to output statistical information about network traffic that had passed through the device. The flow cache has a limited amount of memory available to hold

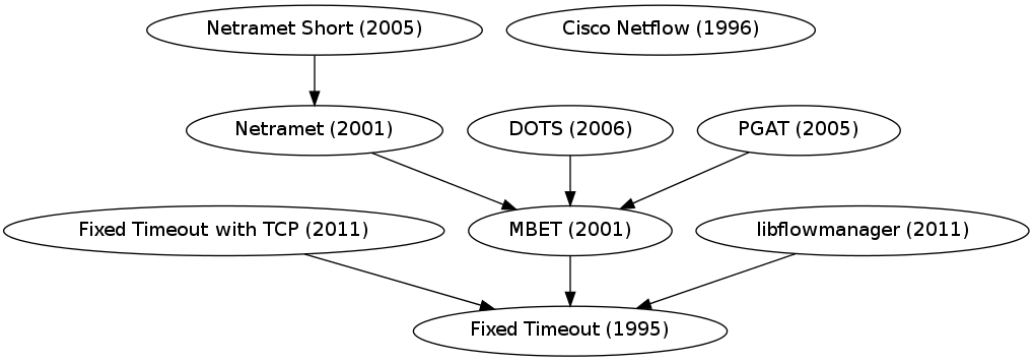

Figure 3.1: Algorithm Evolution Graph

active flows, so the cache is purged of records as soon as flows are deemed not to be active. When a flow entry is expired from the flow cache, a summary record is sent to an external host termed a Collector, which interprets the data, and presents it in a format useful for a network administrator. Expired records are permanently deleted from the Netflow cache.

Flow expiry is determined via a number of methods. If a flow entry has had no activity recorded against it for 15 seconds, the entry is marked as expired. Flow records are also expired if RST or FIN flags are observed in a TCP flow. If the NetFlow cache is full due to a large number of new flow records, the Netflow cache will expire the oldest flows in an effort to keep the cache from running out of memory, whilst keeping the most active records in the cache. When a flow has been active for 30 minutes, and a subsequent packet is received, NetFlow will expire the flow, and create a new record in the cache. According to Kohler (2004), this active expiry mechanism is undertaken to stop counters from wrapping. The counters affected are not specified, but since the maximum value for the Active Timer field is 60 minutes (Cisco, 2007), presumably the field is a 32 bit counter for the packet or traffic byte count for the the flow. The NetFlow specification (Cisco, 2007) does not explicitly include a notification that a flow has been actively expired when a flow is exported, and leaves the role of joining long running flows that have been actively expired to the NetFlow Collector. In a presentation (Kohler, 2004) at the Cisco Networkers conference in 2004, Kohler notes that in terms of memory

usage, an active flow uses 64 bytes of memory per entry, equating to 64,000 entries per 4MB of DRAM memory on the device. He further notes that CPU utilisation adds between 15% to 20% CPU usage when the NetFlow engine is enabled on Cisco switches and routers. A Cisco 6500 router (Cisco, 2007) suitable for a campus environment will normally be delivered with between 128MB and 1024MB of DRAM, depending on the purchased configuration of the supervisor module. The default flow cache value for a 128MB memory module is 128,000 flows.

3.2 Fixed Timeout

Claffy et al. (1995) drew together two ideas - the first of which is that a series of packets between two applications can be uniquely identified by a 5-tuple, which they referred to as a flow. The second is the a concept that any flow, regardless of protocol, could be considered to be expired if the last packet on a flow was more than a fixed timeout of some N seconds prior. Any subsequent packets that contain the same 5-tuple are considered to be unrelated to the previously expired flow, and are created as new flow records. The optimal value for a timeout was determined through experimental analysis of available flow archives, by generating ground truth data for flow records in the flow archives, then testing various timeouts against this ground truth data. The timeout values that were tested were powers of 2, ranging from 2 to 2048 seconds. Lower timeout values caused more flow records to be generated, and higher timeout values caused larger amounts of simultaneous flow records to be held in memory. The authors used a 64 second value to illustrate the differences between a fixed and unlimited timeout model, but while they provide some evidence to suggest that is is a effective value for the trade off between new setup and memory usage, the paper does not explicitly specify that the 64 second value should be a default timeout value for future work. The paper also notes that there are specific problems with this 64

second value for a specific application type: “For telnet flows using a 64-second timeout the median number of packets per flow was 20; with an unlimited timeout this median jumps to 78 packets, suggesting that telnet flows are often idle for more than 64 seconds.” Claffy et al. (1995). This warning appears to have been unheeded. The 64 second value appears in many subsequent papers, including all of the the papers that describe the following algorithms.

3.3 Fixed Timeout TCP Variation

This approach is not one found in the literature, but was added when the Fixed timeout algorithm was implemented. It takes the end of flow references from TCP RST and FIN flags and uses them to mark a flow as expired. This reduces reducing memory allocation by allowing flows that no more traffic is expected to be received on to be closed earlier than they would be under a 64 second timeout model.

3.4 Internet Standards Based Timeout

This timeout mechanism is found in Libflowmanager, as authored by Alcock (2011), and takes its timeout values from various Request for Comments (RFC) documents. RFC documents are used as part of the Internet Standardisation process, which govern how Internet Protocol standards are ratified. The RFC documents that Libflowmanager uses for timeout information are:

- RFC 4787 - Network Address Translation (NAT) Behavioural Requirements for Unicast UDP (2007)
- RFC 1122 - Requirements for Hosts - Communication Layers (1987)
- RFC 5382 - NAT Behavioural Requirements for TCP (2008)

Libflowmanager attempts to model what an operating system would typically do in timing out flows, and uses timeout values from the RFC’s noted

above. The RFC values define a set of timeout values that apply when different conditions are met for a flow. These conditions are based on the protocol of the flow - TCP flows are treated with different timeout values than what UDP flows are. Additionally, the RFC's specify differing timeout values based on the inferred state of a flow. For instance, a flow consisting of a single TCP packet has a vastly different timeout to a TCP flow that has undergone the 3 way handshake.

The specifics of how the various timeouts are applied from the RFC documents follow. Libflowmanager defines an unestablished UDP flow as a flow with one packet, and applies a short timeout of 10 seconds to the flow. If a UDP flow is observed to have more than one packet, Libflowmanager considers the flow to be established, and assigns a longer timeout to it, expiring the flow after 2 minutes of inactivity (Audet and Jennings, 2007). The author notes in the code (Alcock, 2011) that this early timeout value for an unestablished UDP flow is only an experimental technique that has no empirical justification.

Libflowmanager has two modes of operation for creating a flow from an observed TCP packet:

1. Only create a new flow if a SYN flag is seen on the first packet,

2. Create a new flow on any new observed TCP packet.

If the first mode is set, Libflowmanager treats a new TCP flow as an unestablished TCP connection and expired after two times the Maximum Segment Lifetime (MSL) (Braden, 1989). The MSL is defined as 120 seconds, so the timeout is set to 240 seconds, or 4 minutes. If a second SYN-ACK packet is matched to a flow in the unestablished state, the TCP stream is considered to be established by Libflowmanager. Established TCP connections are set with a timeout that will expire them after 2 hours and 4 minutes after the last packet on an established flow has been seen (Guha et al., 2008).

If the second mode is set, new TCP flows initiated without a SYN flag will timeout after 120 seconds of inactivity. TCP flows that start with a SYN

take the same series of timeouts as with mode 1, first being set to a timeout of 120 seconds, and then being set to an established state after the 3-way TCP handshake has been completed. A second packet sets the flow status to be established, with a timeout of 2 hours and 4 minutes. A flow that has one observed FIN flagged packet is considered to be a half closed TCP connection and is expired after two times the Maximum Segment Lifetime (Braden, 1989): 240 seconds. No further changes to the state of a flow are made if any subsequent packets are received. Optionally, TCP sessions that have seen a FIN in both directions can have a 120 second timeout before expiry, but are otherwise expired as soon as possible. If an ICMP packet with an error status is observed, the error message is attempted to be matched to an existing flow, and the flow is marked as closed. Non-error ICMP packets are created as a new flow when the first unique packet is observed, and set with a timeout of 120 seconds. The origin of this value for a timeout is unknown, except for a note in the code stating "Unknown protocol - no sensible state is possible".

This algorithm is occasionally referred to as the "RFC" algorithm in this paper, as Libflowmanager is also the software extended to implement other timeout algorithms.

3.5 MBET

MBET is the common name for the algorithm detailed in the paper "Measurement based Binary Exponential Timeout", by Ryu et al. (2001). This algorithm is designed to match the observed packet rate of a flow to a timeout value. The aim of the algorithm is to change the timeout value based on the observed flow rate of packets in a flow. This is the first of the adaptive algorithms that recalculate the inactivity timeout on a per flow basis. The section introducing this approach is headed as an Adaptive Flow timeout Strategy in the MBET paper.

The initial timeout for a flow is set to either 128 or 64 seconds, and MBET

incrementally reduces the timeout by powers of two, down to 8 or 4 seconds if increasing flow rates are met. The difference in initial and minimum timeout values is due to the six possible configurations available for the algorithm, which specifies flow check intervals, and minimum timeout values.

The 6 sets of available configurations are listed as Appendix 1 of this report, and CFG-3 was the one used when comparing algorithms to each other in the research component of this paper, so is described in detail here. CFG-3 was used in the report as it was the predominant configuration used in the MBET paper, and was used as a configuration option in subsequent papers by Ming-zhong et al. (2006) and Wang et al. (2005) when comparing new timeout algorithms against MBET. All six configurations are included in the implementation of this algorithm, and are able to be selected using a command line argument.

When the MBET algorithm is first run, a series of steps must be undertaken to convert the configuration option into a set of check points for flows to be analysed against. Possible flow timeout values must also be calculated, based on the user selected algorithm.

Program 3.5.1 describes the process of calculating startup values for MBET using CFG-3. The S and To values, along with the P[] array are specified in the configuration.

A listing of all possible configurations is located in Appendix A.

The packet check points are when the timeout values are recalculated. P[] is computationally easier to use when the order in the array is inverted (i.e, the second packet is checked before the seventh).

At each packet check point, the algorithm evaluates if the time between the first packet in the flow and the time of the current packet is less than the current timeout time as set by T[n]. If T[n] is greater than the elapsed time, T[n] is incremented, setting a new lower timeout value to be applied to the flow.

The algorithm then sets the timeout for the flow at the first seen packet to

```
//To compute P[] and T[]:
S   = 5;              //number of timeout values
To  = 4;              //minimum timeout (seconds)
P[] = {7,6,5,4,3,2}; //packet check points
//inverted P[]:
P[] = {2,3,4,5,6,7};
//calculate T[]:
for( int i = S, i >= 0, i-- )
  T[i] = (2 ^ i) * To;
//alternatively, the hardcoded value of T[]:
T[] = {128,64,32,16,8,4};
```

Program 3.5.1: MBET Initialisation for CFG-3.

be T[0]: 128 seconds. The algorithm then sets the next packet checkpoint to be the value of P[0]: 2, and records the time of the first packet in the flow.

The net effect of this algorithm is to rapidly set a static timeout based on the first few packets of the flow, in an attempt to make a adaptive algorithm that sets differing timeouts based on the initial packet rate of a flow. This is an example of an algorithm that uses a restricted set of points from which a timeout value can be recalculated.

3.6 Netramet

Originally published in a paper by Brownlee and Murray (2001), titled Streams, Flows and Torrents, Netramet is an implementation of the aims specified in RFC 2722: Traffic Flow Measurement: Architecture (Brownlee, 1999), which aimed to undertake the following objectives:

- Understanding the behaviour of existing networks
- Planning for network development and expansion
- Quantification of network performance

- Verifying the quality of network service
- Attribution of network usage to users

In the Streams, Flows and Torrents paper, the authors note that the network profile of most UDP transmissions like DNS will look very different to a long running TCP connection. The authors referenced the work by Ryu et al. (2001) in creating the MBET algorithm, noting that Ryu et al. suggested that an adaptive timeout algorithm may be more effective than static timeouts, particularly as most flows appeared to be of a consistent packet rate. This observation was used to create an algorithm that recalculates a timeout based on all packets in a flow. This algorithm diverges from MBET, as MBET only recalculates its flow timeout value based on the first few packets at the beginning of a flow. Netramets adaptive algorithm determines timeout values on a per flow basis by calculating the average inter-packet time value for all observed packets in a flow, and multiplying it by a constant, causing the algorithm to vary its endpoint estimation based entirely on average packet flow rates. Two further flow ending conditions are set for Netramet. If Netramet observes a TCP flow containing a RST or FIN flag, it will use these flags to expire flows before the timeout period for that flow has completed. Netramet has an fixed initial expiry time of 5 seconds for any new flow, which "... allows the meter to make a good initial estimate of the streams interarrival time." (Brownlee and Murray, 2001).

The default suggested value for the Netramet constant multiplier is 20. These values are set from a empirical perspective: "... pragmatically chosen since they work well in practice, achieving reasonably low memory usage for streams, while allowing observation of streams with lifetimes from 10 ms to more than 60 seconds."(Brownlee and Murray, 2001).

3.7 Netramet-Short

A variation on Netramet was proposed, though it does not appear to be used in the standard Netramet distribution (Brownlee, 2005). In this paper, the author suggests that if the total number of TCP packets received for a new flow is less than 6 packets, the flow should be expired after a minimum timeout period. This minimum timeout period is not detailed in the paper, but must be less than or equal to the initial 5 second new flow period for Netramet.

3.8 Dynamic Timeout Strategy (DoTS)

Ming-zhong et al. (2006) set out to generally improve on the MBET algorithm for TCP flows. For non-TCP flows, they suggest using MBET in CFG-3 mode. For TCP flows, the authors apply several different expiry methods, somewhat similar to the RFC timeout methods, in that the timeout is dependant on how long the flow has been active, and how many packets have been observed. Initially, if a flow fits into the category of having less than 6 packets, and its flow duration is less than M seconds, a short timeout expiry of 16 seconds is set. M is recalculated on each new received packet by multiplying the number of packets in the flow by 3.2. If RST or FIN flags are observed on a flow, the flow is set to be expired. Otherwise, a flow is expired based on a longer timeout period of 64 seconds.

3.9 Probability-Guaranteed Adaptive Timeout

In the Probability-Guaranteed Adaptive Timeout paper by Wang et al. (2005), the authors suggest that the ideal timeout strategy would be to predict the arrival time of the next packet in a flow perfectly, so that a flow is not expired early by a short timeout period, and not left with a timeout value longer than needed.

To this end, the authors measured the proportion of maximum packet inter-

arrival times (MPIT) of packets in a flow. They further differentiated their results by splitting the flows into TCP destination ports. Destination ports are the port that a host will initiate a connection to, and tend to be standard values: TCP port 80 is almost always bound to an HTTP server.

The authors analysed a set of captured traffic from Auckland in 2000 and 2001 and observed that as the length of a flow increased in packet count, the MPIT value of a flow was likely to be different to the MPIT value at the beginning of the flow. They further observed that the patterns of MPIT values were different on a per destination TCP port value basis.

The authors accommodate these observations by including a series of recalculation points in the algorithm to recalculate a timeout value for a flow at 10, 100 and 500 packets into the flow. Additionally, the authors create differing timeout values for different destination TCP ports. The authors introduce a probability value into their calculations, allowing the algorithm to be tuned to have shorter timeout values, at a cost of a percentage of flows being split by an early timeout. UDP flows are not covered by the PGAT algorithm, and for these UDP flows, the paper specifies the use of the MBET approach for timeout generation. TCP flows with destination ports not explicitly specified in common ports listed in the PGAT lookup table, located at the end of this document as Appendix B use the "other" set of values at the end of the table. The lookup table for PGAT is included in this document as Appendix B.

In the paper, the authors experimentally replicate the observation made by Claffy et al. (1995), that any timeout value less than the length of the entire trace (or infinity for a live capture) will have a percentage of flows that will be split into two or more flows by an early timeout. The authors of this paper noted this limitation and experimentally determined that the same problem still applied to telnet some four years later. The authors also experimentally found that the 64 second timeout limitation also expired a large number of long running ftp flows. The authors set the maximum timeout any flow timed out by PGAT with a probability of less than 1 to be 64 seconds. Why the authors

did not have a greater range of timeout values not clear from the paper.

For a new flow, the algorithm matches the destination port of a flow and attempts to match it to a named port on the PGAT probability table. If the port is a non-named value, then a default table is used. Using a default probability of 0.8 (80%), the algorithm sets an initial timeout by selecting the 0 packet row, then transitioning along the table until it finds a value that is greater than its probability value.

```
double pgat_lookup[7][4][8] =
{
 { // http lookup table
   // 2       4       8       16      32      64    ∞    (MPIT)
  { 0.6364, 0.7261, 0.7865, 0.8414, 0.9134, 0.9451, 1 }, //  0
  { 0.5955, 0.6956, 0.7642, 0.8289, 0.9150, 0.9495, 1 }, // 10
  { 0.4355, 0.5644, 0.6640, 0.7429, 0.8871, 0.9517, 1 }, //100
  { 0.2959, 0.4145, 0.5114, 0.5911, 0.8230, 0.9248, 1 }  //500
 },
}
```

Data Structure 3.9.1: PGAT HTTP Probability Lookup Table

Data structure 3.9.1 shows the lookup probabilities for a TCP flow with a destination port of 80 (HTTP). Time out values increase on the X axis, and packet check points increase down the Y axis. Comments to the right hand side of the structure indicate at what count the packets will be checked at. The full table can be found in Appendix B.

To calculate the initial timeout of an HTTP flow with a probability of 0.8, at packet 0, then the initial column would be column 3 (first column in the table is considered column 1). The timeout for that flow, for the first 10 packets is then set to be 8 seconds (2^3). If the flow has not timed out and is still active at 10 packets, the timeout value is kept at column 3: 8 seconds (as a probability of 0.8 is greater than 0.7642, but less than 0.8289). At 100

packets, the flow will have a timeout value of 16 seconds, and at 500 packets, a timeout value of 16 seconds.

The drawback of this algorithm is that if the probability is set to anything less than 1, the maximum inter-packet gap that a flow can have is 64 seconds. Lowering the probability further shortens the maximum gap that a flow can have, targeting flows that have large spaces with no traffic in them. Specifying a probability value of 1 would translate to a timeout value of 128 seconds for all flows regardless of packet count or protocol.

3.10 Quan and Carpenter (2011)

An unpublished paper entitled Some Observations on Individual TCP Flows Behaviour in Network Traffic Traces by Qian and Carpenter (2011) specifies that the space for flows in memory to be a very large value (100,000 entries), and only checks to expire flows based on a 30 second timeout if the table is within 95% of being full. TCP FIN flags are used to hint flow expiry for a flow. The authors note that "Our results show that this modification typically allows us to retain a TCP flows entry for more than 10 minutes silence. We believe this is a more accurate method than the traditional fixed timeout method, especially for detecting long-active but sparse TCP flows." (Qian and Carpenter, 2011).

3.11 Discussion

The process of implementing and analysing these algorithms is discussed in Chapter 4: Comparative Research Process. Decoding these algorithms from paper into pseudo-code took far longer than expected due to the terse nature of the academic papers presenting them.

Chapter 4

Comparative Research Process

To compare the algorithms described in Chapter 3, the algorithms had to be decoded from their research papers into pseudo code, implemented, tested for correctness, and then compared in a deterministic fashion using real world data.

4.1 Libflowmanager

Libflowmanager (Alcock, 2011) was used as a template to implement algorithms, as it already had the core infrastructure present around extracting packets from a trace and processing them into a format ready for flow analysis. Additionally, Libflowmanager had the RFC timeout algorithm implemented, useful for using as a template when implementing other timeout algorithms. To process a trace file containing packets of data, Libflowmanager opens a the trace file and reads packets on a per packet basis from the trace. After a packet has been loaded into memory, it will attempt to match the loaded packet to an existing flow. If the packet cannot be matched to an existing flow, it will construct a new flow entry in memory, using the details present in the non-matching packet. After a packet has been processed, an expiry function iterates over the lists of flows. The expiry function then looks to remove any flows that have an expiry time in the past.

4.2 Algorithm Implementation

Algorithms were all initially documented as pseudo code when being decoded from their individual papers. This was undertaken to gain a thorough understanding of how an algorithm worked before implementation of the algorithm in code.

The theory applied here was that re-work of an algorithm would be easier on a whiteboard than in C++. It was anticipated that common patterns would be recognised, and the implementation process could be decoupled from the comprehension process. This was the case; implementation of algorithms by extending Libflowmanager proceeded smoothly, and produced deterministic results when run on trace files. One benefit of this structured process was the realisation that the proper test of an implemented algorithm would be to generate a model of how an algorithm would timeout a series of packets, and then prove the correctness of the implementation by running this series of packets against the then implemented algorithm.

These tests were implemented as a series of files containing packets with explicitly known characteristics. These trace files were then used to validate the implementation correctness of an algorithm, for the expected case as well as what were considered edge cases, such as exceptionally long waits between packets. Additionally, the tests were effective at finding subtle timing bugs that would not have been exposed if a less through approach had been taken. The down side of testing timing with a small number of flows was that bugs that occurred with a large number of flows tended not to be discovered until large traces were run, at which point debugging an algorithm was considerably more complex due to the large amount of data present.

After the psuedocode for all the algorithms were written, Libflowmanager was extended to handle more than one algorithm. Each algorithm was then implemented by creating a new matching and expiry function for an algorithm, by initially duplicating and then modifying the packet matching and expiry code in Libflowmanager. All algorithms apart from the Cisco Netflow and

Quan and Carpenter (2011) were implemented into Libflowmanager. These were left out mostly due to time constraints.

This decision to keep all of the source in the same program, rather than duplicating the code base as individual programs, meant that refactoring the packet matching code back into a series of simpler functions later on in the project became much more effective. This was because the infrastructure that called different matching functions could be condensed back down into a single function call from the initial example program executable, lessening the possibility of changes in one set of matching code not being applied to all the other packet matching code.

A fairly time consuming problem was encountered when using the Standard Template Library (STL) List data structures present in Libflowmanager. These structures are used by the Libflowmanager flow expiry mechanism to expire and delete flows from memory that have an expiry time in the past. Expiry Lists are sorted in insert order and have a O(n) complexity for when the size of the list is requested. In the original RFC algorithm implementation, there are multiple Expiry Lists, used for different states of a flow. One such example of an Expiry List is the TCP Established List, which has a timeout value of 2 hours and 4 minutes. Another example is the UDP Unestablished List, with a timeout of 10 seconds. Each of the Expiry Lists has its own static timeout value.

Any new packet that updates or creates a flow structure in Libflowmanager triggers an update of the timeout for that flow. A flow's timeout value is calculated by adding a static timeout value to the time on the latest packet to arrive. The particular Expiry List that a flow sits on is then reordered by moving the flow with the new timeout value to the tail of the list, indirectly ordering the list by shortest to longest timeout.

After a packet has been processed into a flow and the timeout value recalculated, the flow expiry mechanism is run. The expiry mechanism iterates through each Expiry List from the head of each Expiry List where flows with

the shortest timeouts are. If the first flow at the head of an Expiry List does not need expiring (its timeout value is in the future), the expiry mechanism moves on to another Expiry List.

This was a subtle bug to find and fix, because the expiry mechanism only marks a flow as expired, removes the flow from memory and writes it out to disk. Flows are not otherwise directly modified by the expiry mechanism. A flow not expired until the program ends due to this bug will have a correct timeout value set, but would still be marked as active instead of expired. This was easy to miss since even the smallest trace contained several thousand flows.

The problem was that only fixed timeout mechanisms like the Fixed Timeout and the Internet Standards Based Timeout have static values for expiry times. Other algorithms, like MBET and Netramet vary the timeout value that is applied to a flow on a regular basis. This means that an algorithm with a dynamic timeout value will cause flows in a standard expiry list to be become unordered.

The increase of unexpired flows in memory meant that when a 800MB trace was run, execution time for Libflowmanager running the Netramet algorithm ballooned out to two days. With around 90 days left in the project this meant that a maximum of 45 files could be analysed. A request was made to gain access to the Symphony Cluster at the University of Waikato.

Memory and CPU profiling was undertaken with the VTune suite from Intel (Intel, 2011). This was used to run performance analysis on Libflowmanager running a dynamic algorithm, and showed 98 billion transactions on a list.size() call, compared to 6 million on the next most utilised function. Examination of the STL template library documentation revealed a note at the bottom of the reference page for the list.size() call that noted that the function was “Linear in some implementations” (cplusplus.com, 2011b). Further instrumenting the expiry list to run a large trace then stop half way through and print out the contents of the Expiry Lists revealed the extent of lack of ordering when dynamic timeout algorithms were being run. The solution was

to create another set of expiry lists, using the Multiset (cplusplus.com, 2011a) list structure, which allowed a custom sort function to be defined so that a flow could be inserted into a Multiset based on its newly calculated timeout value. This new list structure was then retrofitted to the rest of the dynamic timeout algorithms. This proved to be a significant piece of work, as some of the management structures depended specifically on a List data structure being used. Currently, Multiset data structures have been implemented for all dynamic timeout algorithms, but have not yet been extended into the static timeout algorithm implementations. Since this was a complex change to the core of some of the program code, a series of unit tests were created to verify that the new Multiset based Expiry Lists are ordered as expected after flow insertion, deletion, and modification. These unit tests currently all pass, are incorporated into the main program, and can be run from the command line by specifying the '-t' switch.

The expiry mechanism of Libflowmanager was extended to write a standardised format line into a file when expiring a flow, rather just than deleting the flow from memory. This meant that when the output of algorithms was to be compared against each other, Libflowmanager output could be read in from files in a standard format, and processed by a programming language suited to text processing. The details of the comparison script are discussed in Section 4.5, Post Processing. The next section details the data and the methods used to generate comparison files that were used by the scripts detailed in the Post Processing section.

4.3 Real World Data

The second problem of a deterministic set of working data was solved by utilising the WITS archive (WAND, 2011).

The WITS archive contains a very large collection of anonymised captured network traffic. Given that the volume of trace data stored in the WITS archive

approaches 100TB, one or more dedicated machines on campus were needed to be used to process data. Additionally, any trace that was to be processed would have to be run by nine different algorithms. Using the Symphony Cluster to run traces in parallel on any tracefile while utilising Symphony's the high speed network connection to the WITS archive was an obvious solution to this problem.

Since the algorithms are all implemented in the one instance of Libflow-manager, command line switches were added during the development process to switch between the various algorithms. This functionality was built on to generate a series of output files for all algorithms on a single trace file.

Initially, this 'runall' script was used to run all algorithms against a tracefile on a single CPU host. Further work extended this script to submit the jobs onto the symphony cluster for processing in parallel.

4.4 Symphony Cluster

The 'runall' script in the previous section was rewritten to submit jobs on to the cluster. It took a single argument - a tracefile, and took care of the rest of the submit process for running all algorithms on the cluster. Further work eventually extended this script to submit the post processing script onto the cluster as well, but added dependency checking to the final job so that it only ran if all the algorithm execution jobs for that trace had successfully finished.

Figure 4.1 describes the flow of submitted jobs through the cluster. The dashed red lines represent jobs being started and completed on client machines by the queuing software running on the Symphony Cluster. Nodes marked with a tan colour mark stages of the process that involve reading and writing of data over the network from a cluster client machine to the master (also known as head) cluster control node. This network traffic was generated by a Network File System (NFS) mount point, and occurred when any of the processes read and wrote data to the /home directory, hosted on the head node, hn1.

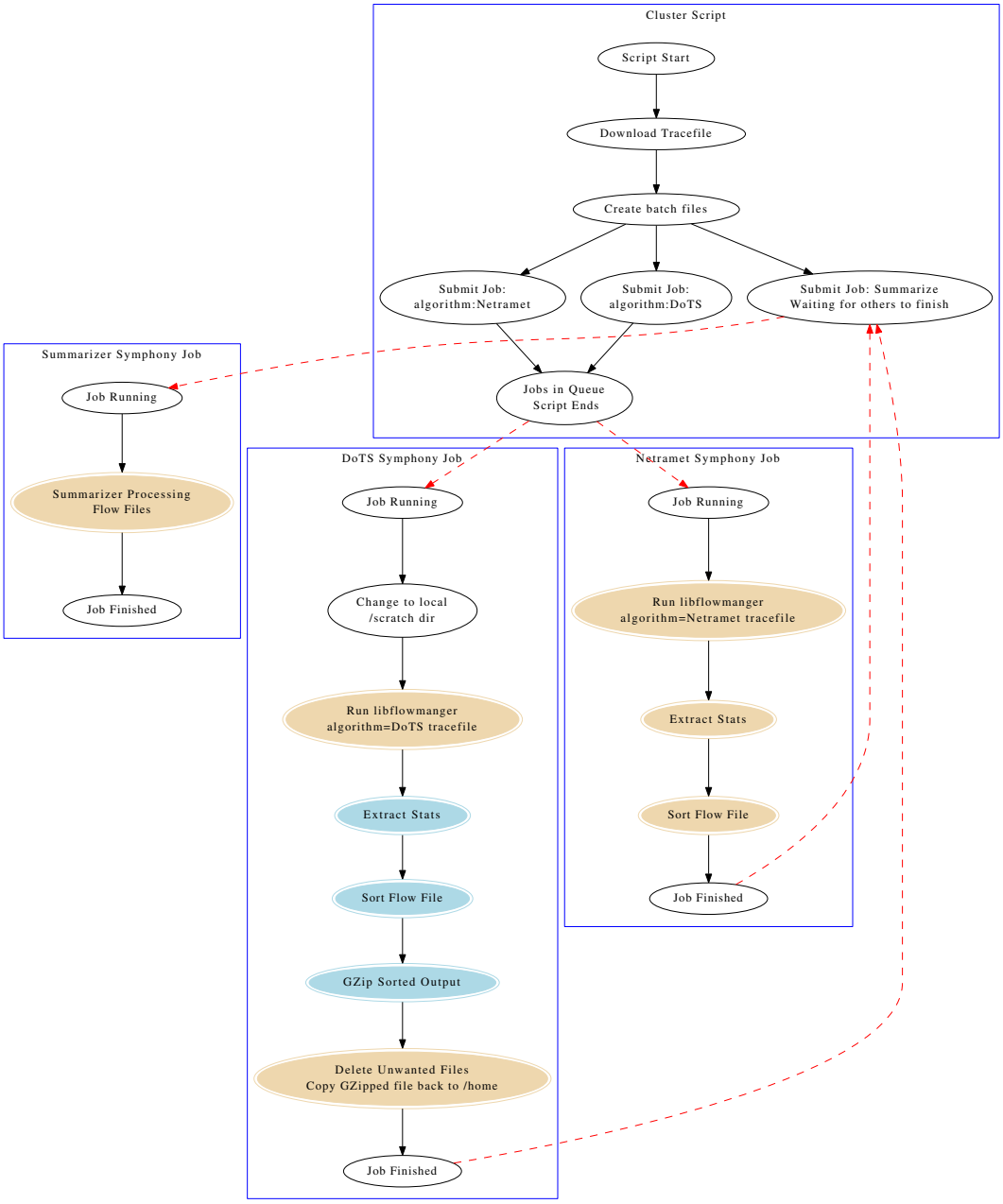

Figure 4.1: Cluster Program Flow

All the machines making up the Symphony Cluster have Gigabit Ethernet connections to a central switch, and cluster machines running Libflowmanager jobs on the network swamped the network link attempting to read and write multi-gigabyte output files over NFS to the single network link to the hn1 head node on the cluster. The Netramet Symphony job in the right hand blue box in figure 4.1 shows the original configuration for a cluster job run, and the middle box in figure 4.1 shows the reworked run of a cluster job, designed to minimise cluster network traffic back to the head node. Blue nodes indicate local operations on the client machine. A further optimisation of the final write back step was to GZip the output file before writing it back to the head node. The rationale for this is further detailed in section 4.5.2.

By removing almost all of these read and write operations over the network, the processing speed of traces has increased to the point where a 2 gigabyte trace file can be processed in parallel on the Symphony Cluster with 9 instances of Libflowmanager running, taking about 20 minutes to complete.

To put some perspective on the data volumes, prior to the compression and data reduction optimisation steps being implemented, a typical run of a day's worth of traces from the Waikato V (WAND, 2011) trace set for June 14, 2007 resulted in a final working directory size of 44GB. The original trace size loaded off the WITS archive for this day was 3267MB. A final working directory size for a script running with the gzipped optimisation was 5.6GB, a significant improvement. At the end of the project, with two sets of several week long traces being run on the cluster, the home directory size on the cluster was 1.5 TB, out of a possible 5TB of total drive space for the home directory.

4.5 Post Processing

A method of comparing the output files of the Libflowmanager runs for each algorithm was needed. The output file format is a simple Comma Separated Value (CSV) text file, and is labelled with the name of the trace, the date the

algorithm was run in Libflowmanager, along with the name of the algorithm that the output relates to.

All algorithms needed be compared against each other on a structured basis. Fundamentally, this meant extracting the lines that matched the same flow identifier (5-tuple) from each CSV file, and comparing the output of each algorithm for that flow record.

The storage is laid out in the hierarchical structure described in Figure 4.2.

```
flow identifier → algorithm →|
        |→ flow-record → flow-data-name → flow-data-value
```

Figure 4.2: Data Hierarchy

The flow identifier in Figure 4.2 is a 5-tuple value, and is a simple text format, for example: "10.0.0.1:1234 → 10.0.0.2:80 (TCP)". An algorithm is one from the list documented in the Algorithms chapter. A flow record equates to a line in a CSV file, written out to disk by Libflowmanager. Finally, the flow data name and flow data values relate to inherent properties of that flow record, for example, the start and estimated end times of a flow.

The → indicators in figure 4.2 can be read as a "has" or "have": A flow identifier has algorithms, algorithms have flow records, flow records generated by Libflowmanger have detail names, which in turn have values.

Figure 4.5.1 describes the data structure as implemented. The outer most item is a Perl hash (named %storage) that holds all the flow records and data. Perl has a method of associating data into key = value structures called hashes. These are where a key is associated with a value. Perl extends this by allowing the value to be another data structure, like another hash, or a list. This structure uses these concepts liberally, as demonstrated in Data Structure 4.5.1.

Because of this hierarchy, it is possible to have Perl traverse down the data structure in a programmatic fashion, and determine that 'flow identifier 1' for 'algorithm 1' has a 'flow start' value of 550.

Furthermore, it is possible for the summariser script to compare the two ‘flow expiry’ values for ‘algorithm 1’ and ‘algorithm 2’, and determine that ‘algorithm 1’ estimated the expiry of a flow for ‘flow identifier 1’ some 40 seconds earlier than ‘algorithm 2’ did.

```
%storage = {
    ‘flow identifier 1’ => {
        ‘algorithm 1’ => [
            {   ‘flow start’  => ‘550’,
                ‘flow expiry’ => ‘560’,  }
        ]
        ‘algorithm 2’ => [
            {   ‘flow start’  => ‘550’,
                ‘flow expiry’ => ‘600’,  }
        ]
    },
    ‘flow identifier 2’ => {
        (like above)
    }
}
```

Data Structure 4.5.1: Stylised Perl Data Storage Design

An initial approach to processing the CSV output files generated by Libflow-manager was to load all flows into Data Structure 4.5.1, and process the stored flows afterwards. This approach quickly proved unfeasible. Before being stopped, a script run of 5 algorithms, each with 500MB of CSV files, used 80% of the ram in a machine containing 32GB of RAM in 2 minutes of run time. The script was stopped because other running processes were starting to be pushed into swap, and the script had not moved significantly beyond the second of the five CSV files.

Given that the traces that were to be used later on in the project were likely

to be significantly larger, along with the output from 9 algorithms running on a trace needing to be processed, a large amount of effort was spent on refactoring, making the processing script run as quickly as possible and to use as little memory as possible.

Part of the solution to this problem was accomplished by adding a hashing function at the Libflowmanager stage, and including it as a field of the flow line when it was written out to a CSV file. These CSV files were then sorted on the hash value, using the Unix command line sort function.

This sort mechanism was used due to an observation that sorting by hash invariably takes the order of the flows out of the order that Libflowmanager has written them to a file. The realisation was that this reordering is not important - flows are fully independent from each other, and the order in which they are processed does not matter. What is important is that the processing script can make the assumption that if it has a reached a value for a hash in a file, then the script can assume that it has read any hash of a lower value into memory.

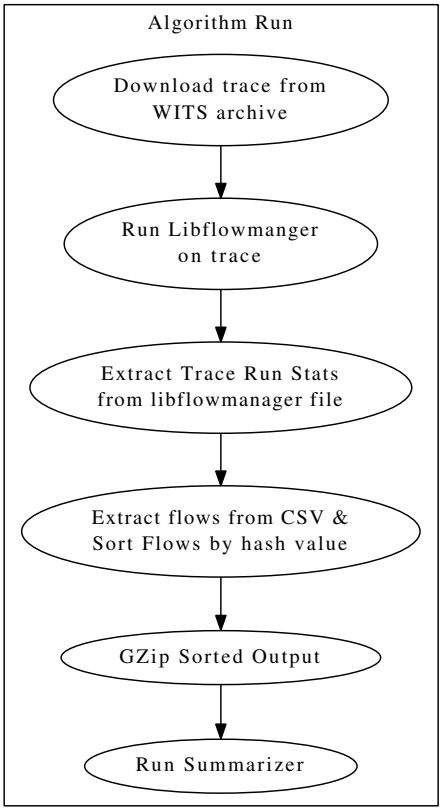

Figure 4.3: Single Job Run

Figure 4.3 describes the typical flow of a trace file being processed by Libflowmanager. This then details the processing of the Libflowmanager output files into a format that the summariser script can load in an efficient manner.

Figure 4.4 describes the program flow of the summariser script. The script reads the same flow identifier from each of the algorithms' sorted flow files, loads the records into a internal data structure, and then processes the records for differences and similarities between the algorithms. Once the record has

been processed, the script deletes the flow records from memory and moves onto the next flow record. If the end of the compressed flow files is reached, the script writes out the summarized results data to disk and exits.

The sorted flow file is read line by line into a Perl script that then converts each line into a part of a Perl hash, indexed by a string representing a flow - not the hash. Keying the Perl hash with a flow string rather than the hash generated by Libflowmanager was a fortuitous design decision, because the hashing function used (Hsieh, 2008) had been observed to regularly have a number of collisions when faced with trace files larger than 250MB. When more than one flow is seen with the same hash, the flows are loaded into memory with different flow keys separating them from each other.

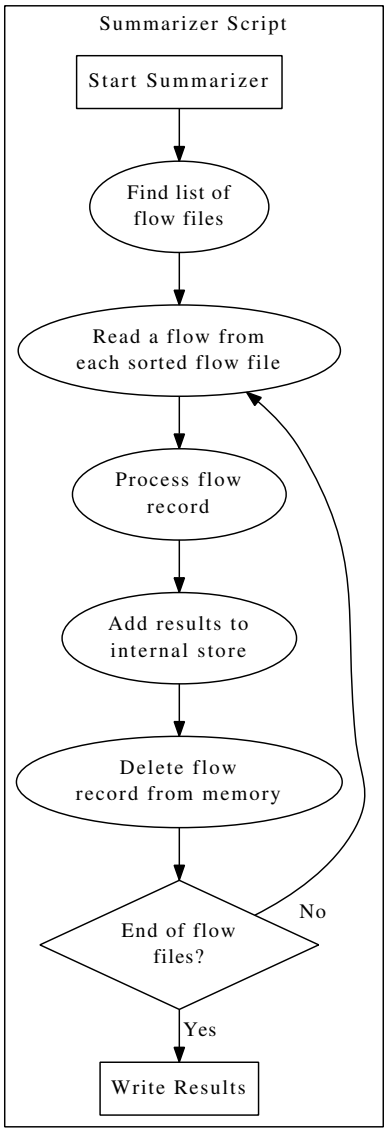

Figure 4.4: Summariser Internals

Structure 3.1 demonstrates a how two algorithm outputs with the same hash and flow value were stored in memory when the Perl post processing script was running. The output of the Perl data storage is generated by the Data::Dumper module (Sarathy, 2010).

On processing a flow, the script looks to check that all algorithms are present for a flow entry in memory. If not, it adds a counter to a general broken flows counter, writes out offending flow to the logs, and continues on. This problem of not all flows having been read in from files into memory has been an area of persistent re-work.

The underlying problem is that the only way of knowing that a hash has been fully read in through all of the CSV files is to load a greater hash value

```
'10.2.145.38:60223->10.1.52.40:21 [tcp]' => {
'mbet' => [ {
            'time_start'  => '550.016076',
            'time_end'    => '550.016076',
            'time_expire' => '678.016076',
            'data_in'     => '60',
            'data_out'    => '0',
            'packet_in'   => '1',
            'packet_out'  => '0',
            'lengthened'  => '0',
            'status'      => 'Expired',
            'list'        => 'Delete List'
            'hash'        => 4294963314,
    } ],
    'claffy' => [ {
            'time_start'  => '550.016076',
            'time_end'    => '550.016076',
            'time_expire' => '614.016076',
            'data_in'     => '60',
            'data_out'    => '0',
            'packet_in'   => '1',
            'packet_out'  => '0',
            'lengthened'  => '0',
            'status'      => 'Expired',
            'list'        => 'Delete List'
            'hash'        => 4294963314,
    } ],
}
```

Data Structure 4.5.2: summariser Storage Hash with Real World Data

in the sorted file. This process tended to partially load a new flow record into memory, and would add to the counter of broken flows. The unsorted retrieval method for a hash in Perl is not related to insert order, so the processing of flows not fully loaded into memory occurred regularly. The fix to this problem was two fold: Firstly, the realisation that a Fixed timeout implementation with a very high timeout (referred to as claffy_unlimited, or unlimited) will have the smallest possible number of flow entries for a trace - only one per flow. Secondly, the the seek function was used to move the file pointer in a sorted Libflowmanager file back one line when an unwanted line with hash value is read. This is done so that the next time a line is read from the same file, it is the first line of that hash value.

4.5.1 Broken and Uneven Flows

After the summariser script was fully debugged and returning consistent results, there was an incidence of broken and uneven flows. Broken flows are where all algorithms are present for the flow identifier, but have missed some packets. Uneven flows are where an algorithm is missing from a flow identifier. For the Waikato V tracefile on the 2007-06-06, there were 9.4 million flows processed. Of these flows processed, one 1 uneven flow result and 5 broken flow results were recorded.

The reasons for why this occurs are not fully understood. What is known at this point is that the bug exists in Libflowmanager, occurs in a deterministic manner, (in that the same trace file run with the same algorithm will generate the same errors) and that if the flow is extracted from the trace file and run through Libflowmanager by itself on all algorithms, the error does not occur. This means that at some point, a series of packets destined to a flow are not attached and processed on that flow. This bug was only discovered after the summariser script was written and thoroughly debugged, which occurred towards the end of this project.

This bug would be investigated further by finding a packet from affected

flow that was not processed, then using that packet as a starting point for enabling debugging output for that particular flow in Libflowmanager. This should show why that particular packet is not being picked up by a particular algorithm. This has not been undertaken, because it only appears to affect a very small number of flows, and at that point the project practical component needed to be stopped, and the final write-up begun.

4.5.2 Further Performance Work

As discussed in section 4.4, performance testing revealed that the network bandwidth used to process the flows was very large, so a piece of re-work resulted in the files being gzipped before being transferred back to the head node. The Perl module for Gunzip (Marquess, 2011), does not seek backwards, so the use of the seek function was replaced with a single local buffer that gets populated when the seek function moves forward one too many lines. This approach appears to be relatively bug free, though the use of Gunzip has doubled the run time of the Post Processing script to 2 hours for 22GB of sorted CSV output files, summarising 9 algorithms on one day worth of traces.

This is a backwards step in terms of runtime for the summariser script, but means that the network is not severely overloaded when traces are running. This essentially replaces a network bandwidth intensive step for a CPU intensive step. Given that client cluster machines have at least two cores, scaling CPU performance by reworking implementations is more effective than scaling network performance by further compressing the data.

4.6 Statistics

This section details the processing function of the summariser script. At a high level, this function creates general per trace statistics, as well as per algorithm statistics from the Libflowmanager output.

4.6.1 General Trace Statistics

The summariser produces the following general per trace measurements:

4.6.1.1 Memory Time and Total Flows

The memory time metric represents the smallest possible time that an algorithm can have all of its flows in memory, given that a flow is defined as starting at the first observed packet, and ending at the last observed packet. This is the sum of all of flows in a trace for the value of the last seen packet time minus the first seen packet time for each flow. The memory time value is generated from a modified version of the Fixed timeout algorithm implementation, with the timeout functions set to a timeout longer than that of the trace.

The Total Flows value is derived from a Libflowmanager, where a counter is incremented every time a new flow is allocated in memory. This is written to disk when Libflowmanager exits.

4.6.1.2 Broken and Uneven flows

This is the counter of the incidents referenced in section 4.5.1: Broken and Uneven Flows. These occur where an algorithm running in Libflowmanager has incorrectly processed a tracefile and missed a packet (Broken Flow) or a Flow (Uneven Flow).

4.6.2 Algorithm Specific Statistics

The processing function further splits the statistics into per algorithm counters, and outputs the following values:

4.6.2.1 Time Wasted

This value is derived from the expiry time minus the last packet time, summed across all flow records in a tracefile. This represents the amount of time that an algorithm has spent waiting for a packet that never arrived.

4.6.2.2 Shortflows

The term "shortening" is taken from the MBET (Ryu et al., 2001) paper and is used to describe a flow timed out by an algorithm before the flow actually ended in the trace.

This type of event causes additional flow records for the same flow identifier to be created in memory and then written out to disk by Libflowmanager. For an accounting application, shortening may be a useful attribute - a flow that has long spaces of no activity in it could be written out to disk, freeing up memory for other currently active flows. This approach is more problematic for a firewall application, since an algorithm deciding to terminate a connection would end up closing a long running application connection too early.

This counter is only incremented once per flow if a shortening event is seen occurring on a flow.

4.6.2.3 Shortened

This is a counter of the total number of Shortflow events, but is incremented for every shortening event that occurs.

4.6.2.4 Lengthening

The inverse of shortening is termed lengthening. This is where a SYN packet is seen in the middle of what is considered to be an established TCP flow.

Generally, the underlying cause of this event is due to the ports and IP addresses unique to a TCP session being reused after a FIN/ACK packet, and the session not being closed by an algorithm that does not read TCP flags for hints to end of flow information.

This poses a problem - the session may not the same as the previous one, and fundamentally, the two sets of data should not be joined together. If this is not taken into account, it can affect the estimation of the time required to leave the firewall ports open, possibly leaving a connection open too long, and opening a machine up to an unexpected attack vector.

The lengthened counter is incremented when a set of SYN packets are seen inside a established TCP flow. The current method uses an approximation, whereby the counter is incremented if a packet is a SYN flagged packet, there have been more than 5 packets have been seen in each direction, the protocol is TCP and a SYN packet has previously been seen in both directions.

This approximation has been arrived at because while the observer in the middle of two peers may see all packets between the peers, data loss between a packet seen by the observer and the destination peer means that the state assumed by the observer may not be that of the peers. Being able to determine exactly when a TCP session is established and not in the final stages of SYNACK conversation with severe data loss are can be quite difficult. Attempts to mitigate around this problem by tracking TCP sequence numbers were hamstrung by a time constraints and a bug where expected values would be occasionally be moved by 2$\hat{2}$4, or $\tilde{1}$6 million. The source of this was not discovered, and in the interests of finishing the project, an approximation was used.

4.6.2.5 High Water Mark

This is a counter generated from the maximum number of simultaneous flows in memory in Libflowmanager. Ideally, a algorithm should hold as few simultaneous flows in memory as possible. The maximum number of simultaneous flows in memory essentially determines the memory requirement for a machine implementing this flow timeout scheme.

4.7 Discussion

This chapter has detailed the implementation design, along with some of the successes and challenges encountered. This part of the project ran from the middle of 2011 until around the middle of January 2012. The next chapter describes the results generated from the above implementation.

Chapter 5

Results

This project looks to evaluate algorithm results from two perspectives: that of a firewall, and that of an traffic accounting server, and seeks to compare algorithm responses. The project also looks to see if changes in traffic behaviour over time have resulted in changes to algorithm responses.

To compare algorithms across time, the Waikato V and the Auckland VIII trace sets from the WITS archive (WAND, 2011) have been used. These are multi-day traces, and are split by four years - the Auckland VIII traceset was captured in December 2003, and the Waikato V traceset was captured between June 2007 and September 2007. The Auckland VIII trace set is only available in hour long traces, whereas the Waikato V trace set is only available in 24 hour long traces. To remedy this, the Auckland trace set was combined into day long traces, then used for further analysis. Tuesday, the 2nd of December, 2003, was a trace of the headers of 58GB worth of data that transited the Auckland link and was comprised of 149 million packets. Compared to the Waikato trace set from Wednesday, the 6th of June, 2007, the Waikato trace captured the headers from 108GB worth of data, and 206 million packets.

It should be noted that the packet capture times, while being a 24 hour period, do not cover the same time range: The Waikato trace set covers the midnight to midnight in UTC, whereas the Auckland trace period covers the midnight to midnight period in NZDT. This detail was not obvious during

the testing phase of the project, and was not accounted for. Another detail missed during the testing phase of the project was that the Waikato traceset is occasionally split into 12 hour periods, being stopped at midnight. It was assumed that all the traces were 24 hour traces. Dates affected were the 9th, 16th, 23rd and the 30th of June, 2007.

A total of 39 individual days of traces were run for analysis on the Symphony cluster. Each of these days were run 9 times - one for a reference trace, where an unlimited timeout was set, and 8 others for each of the implemented algorithms. As noted in Section 4.3, a typical run for a several gigabyte trace file would take about 20 minutes to complete all of the Libflowmanager instances, followed by a 2 hour run time for the summariser.

5.1 Accounting Results

Firstly, we look at the perspective of an Accounting Service, and look at the memory usage of a trace, and ask if the results for a day are representative across a month, and if these results are then representative of differing traces in time. Given that an Accounting Service needs to process vast numbers of flows, the primary goal of an algorithm in this environment must be limiting memory usage. This section looks to quantify the effectiveness of algorithms, by looking at two variables: Maximum memory usage, and memory time.

Maximum memory usage is simply a high water mark of the largest amount of flows present in memory at any one time when an algorithm is processing a trace file. Memory time is based on the observation that an ideal accounting algorithm should look to close a flow as soon as possible after the last packet has been seen. This is measured by evaluating how many seconds in total an algorithm has had all flows open in a trace. Ideally, both of these values should be as low as possible.

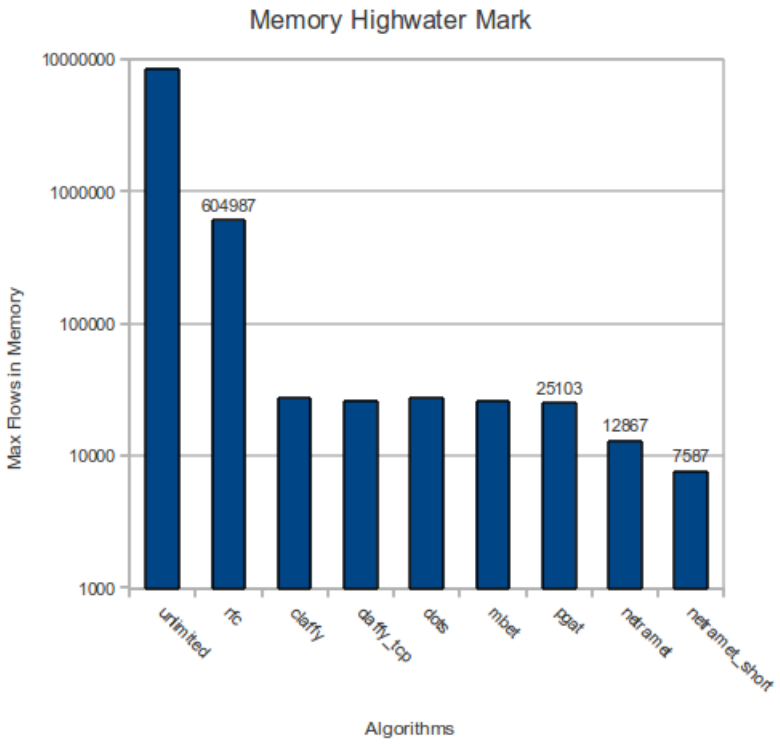

Figure 5.1: Maximum Flows in Memory by Algorithm. From the Waikato V Trace Set on Friday the 8th of June, 2007.

5.1.1 Memory Highwater Mark

Figure 5.1 displays the memory high watermark values for a trace file from the WITS archive, from the Waikato V traces for the day of the 6th of August, 2007.

The X axis lists algorithms ordered roughly by Y axis value and algorithm type. Algorithm types transit from no timeout, on the left hand side, to fixed timeout and then onto to adaptive timeout schemes on the right hand side of the X axis. The Y axis value is the maximum number of flows in memory that was recorded when Libflowmanager was running on a particular algorithm, and is a Log base 10 scale.

There are several observations to be made: Any timeout at all decreases memory usage by at least an order of magnitude. With two notable exceptions, the successors to the Claffy Fixed 64 second timeout have not made any significant impact on memory usage. This may explain why a simple timeout figure, as is the case in many modern contexts, has not been significantly

changed from a fixed timeout scheme where an algorithm is running in an environment where memory usage is the most important factor. As labelled in the graph in figure 5.1, Netramet uses 51%, and Netramet short uses 30% of the memory compared to other timeout algorithms. If a firewall implements timeouts according to RFC derived timeout values, it very well may have resource starvation issues, and be able to only support 4.5% of the flows that a Fixed Timeout algorithm with a 64 second timeout does.

It is possible that these results are dependent on traffic on the 8th of June, 2007, and could vary from day to day. To investigate the stability of the results in Figure 5.1, two more graphs were prepared. The first graph is the rest of the month of 5.2, the 6th to the 30th of June, 2007 from the same Waikato V traceset. A second graph spans the dates from the 1st to the 15th of December, 2003, from the Auckland VIII traceset.

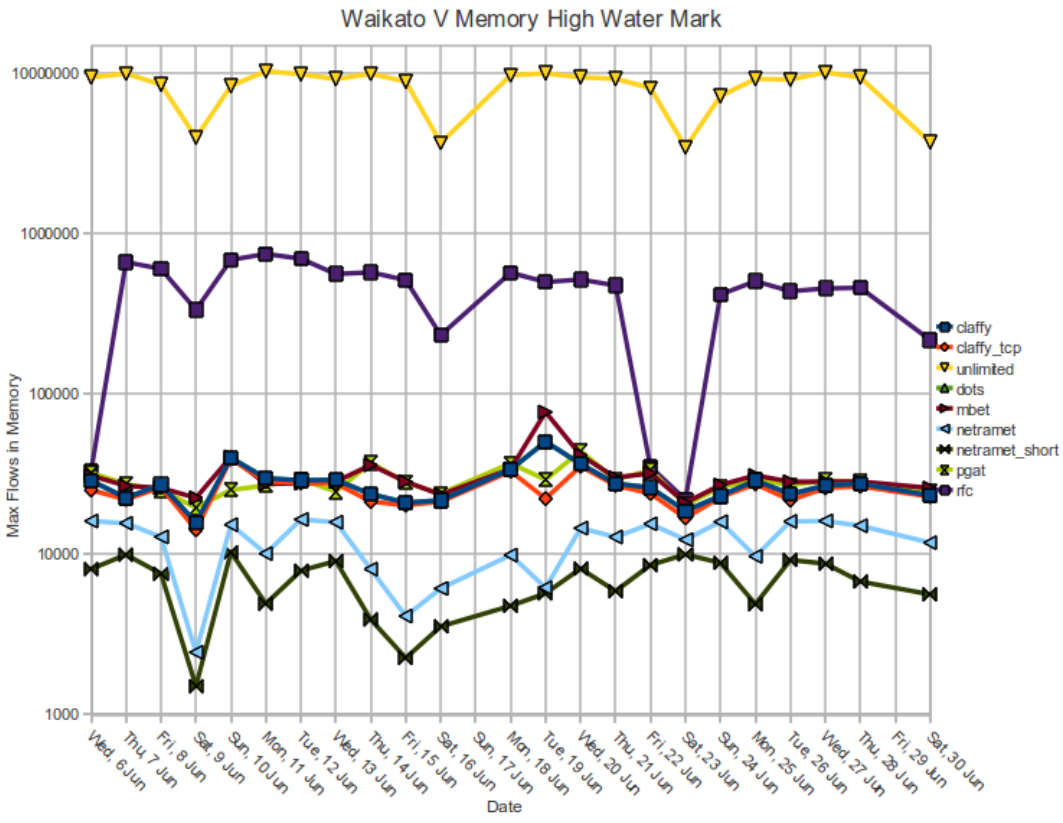

Figure 5.2: Waikato V High Water Flow Mark for Memory Usage

Figure 5.2 is a graph of a 24 days of memory high watermark flow values, we can say that the results from figure 5.1 are indicative of a larger trend - that

Netramet is far more effective than other flow termination algorithms, and that the algorithms are reasonably consistently ranked in terms of effectiveness.

For comparative purposes, and to see if the memory usage behaviour has changed over time, Figure 5.3 is a graph of memory usage from the Auckland VIII trace set, from December 2003. As with Figure 5.2, the X scale indicates the algorithm, and the Y scale is the maximum number of flows in memory that Libflowmanager encountered when running traces on a particular algorithm. The Y axis scales have deliberately been kept at the same values, so that page by page comparisons can be made.

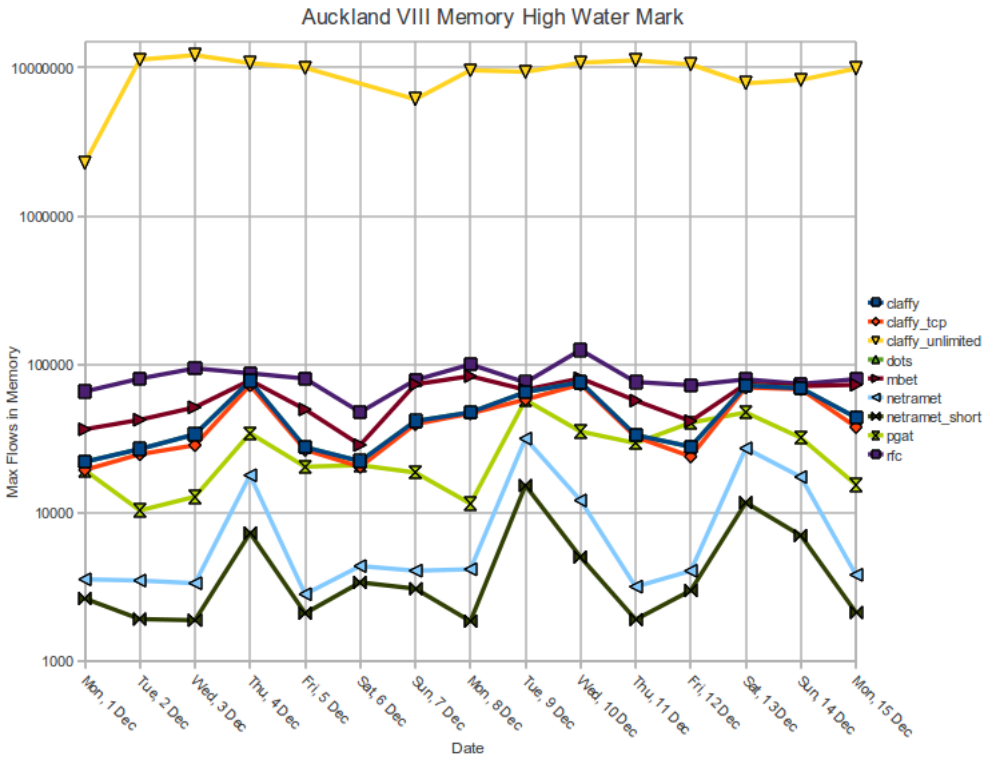

Figure 5.3: Auckland VIII High Water Flow Mark for Memory Usage

Why the RFC algorithm has significantly different memory profile to 2007 is unknown. Factors that affect the time that a flow will stay in memory for the RFC algorithm are TCP FIN and RST flagged packets that cause a flow to be closed and deleted earlier than they would be if they were timed out, and established TCP connections, which cause flows to in memory for 2 hours and 4 minutes. If established TCP connections were being more effectively managed by end hosts in 2003, then this could account for some of the differences.

5.1.2 Total Flow Memory Time

As noted in the introduction to this section, the number of seconds that an algorithm holds a flow in memory relates to the general memory efficiency of the algorithm, but also indicates if an algorithm is likely to close a flow early. Ideally, an algorithm designed for an accounting scenario would have a very low value for this metric, as recording traffic counts about all data observed passing past an observation point is more important than keeping a flow record in memory.

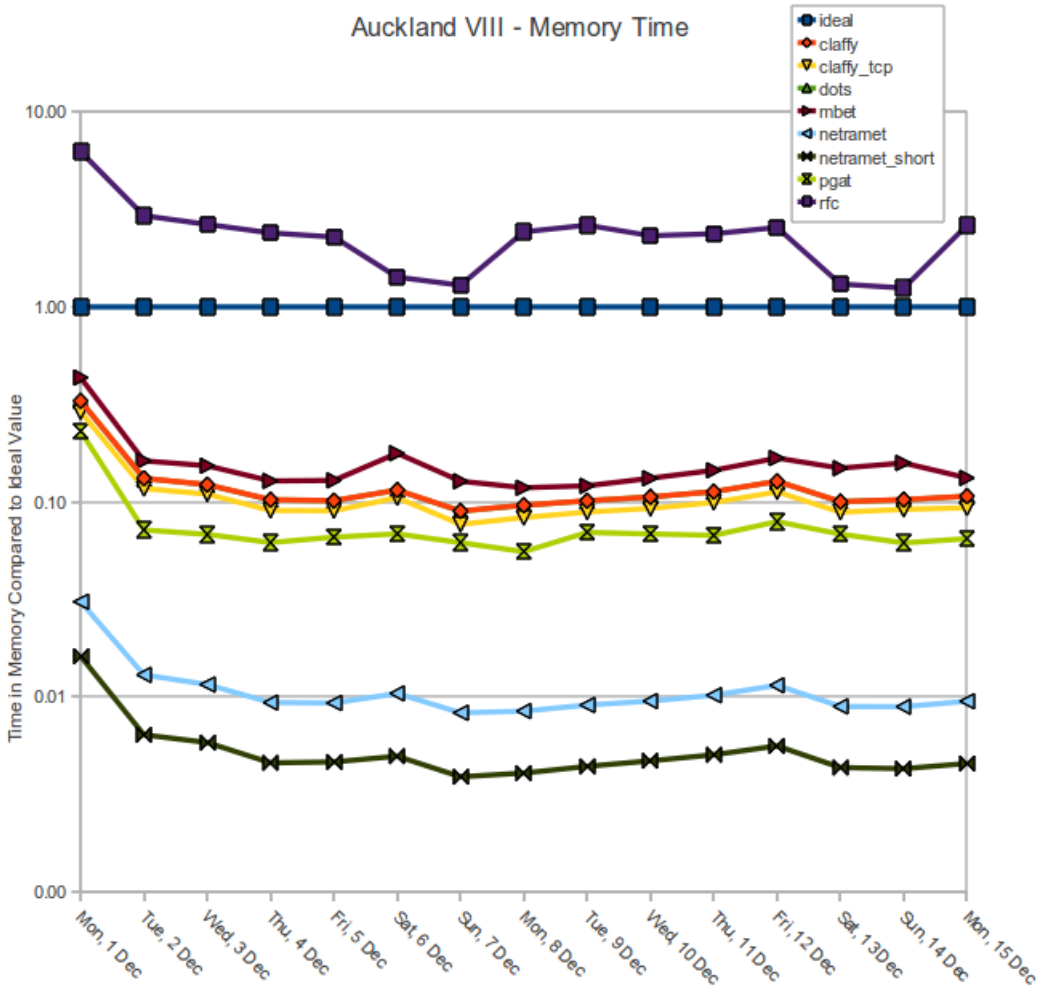

Figure 5.4: Auckland VIII Total Flow Memory Time

Figure 5.4 describes the total flow memory time for the Auckland VIII trace set from 1st of December 2003, to the 15th of December, 2003. Libflowmanager runs were of traces with a length of a 24 hour period, and trace dates are listed on the X axis.

The Y axis values are ratios of the ideal daily total flow memory time to other raw algorithm values. This was undertaken to normalise the weekly variation in memory time - weekends will tend to have less traffic, than week days, which translates directly into smaller values for total flow memory time.

Of note in this graph is that all of the algorithms, except for the RFC algorithm hold flows in memory less time than they actually exist as valid flows in the trace. This indicates that while this is a mark of an effective algorithm for an Accounting Service, there are likely to be problems with shortening of flows with other algorithms when viewed in a Firewall Context.

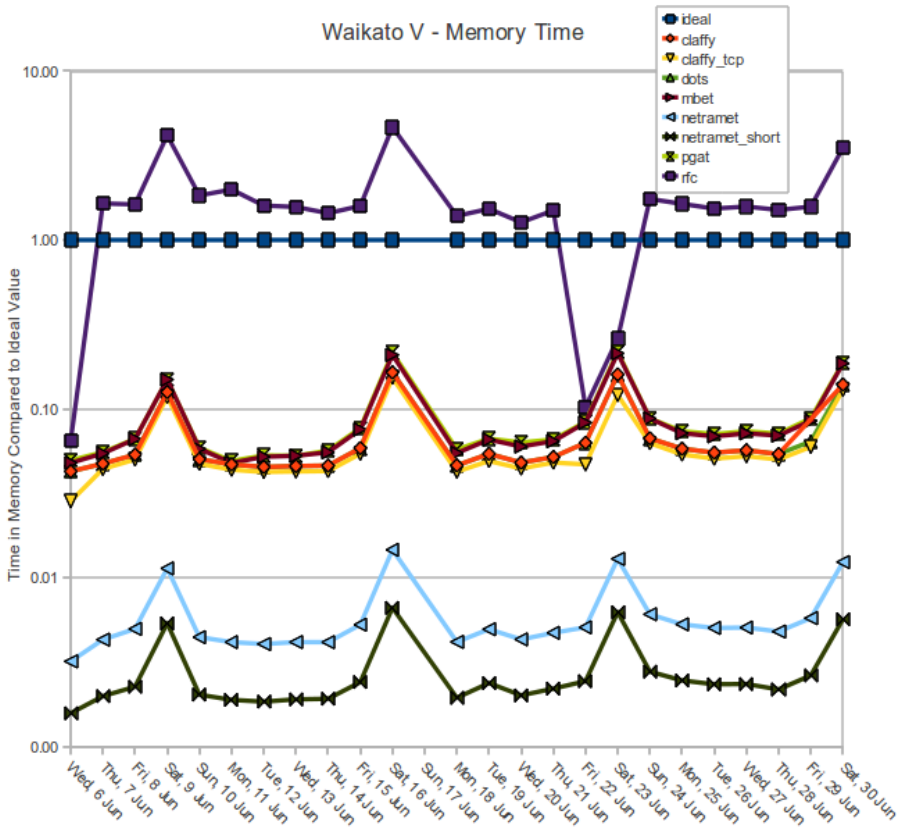

Figure 5.5: Waikato V Total Flow Memory Time

Figure 5.5 has the same X and Y axis scale characteristics as 5.4.

Netramet is an order of magnitude more efficient than the other algorithms at keeping flows out of memory, making it an effective traffic accounting algorithm. There also appears to be slight increase in efficiency between the two time periods, where all algorithms tend to be sitting a bit lower in 2007 than they were in 2003, though this may be due to any number of external factors,

such as seasonal variation or intrinsic site differences.

5.2 Firewall Results

When evaluating algorithms in the context of a Firewall Application, a useful metric is shortening. This is where an algorithm has made a decision to close a flow in a trace before all the packets have been received for that flow identifier. A firewall that closes a flow too early will cause end hosts to experience either unexpected packet loss because a firewall has decided to drop an unwanted packet, or a connection termination event, where a packet gets rejected by a firewall. This behaviour will cause unexpected events for users, who will notice their connections undergoing arbitrary disconnections after a short period of inactivity.

Figure 5.6 is a count of any flow that experienced shortening. The X axis lists the trace dates, and the Y axis lists the relative number of flows shortened, where a relative value of 1.00 on the Y axis would indicate that all flows had been shortened by an algorithm in that 24 hour period captured by the analysed trace file.

Using a ratio means that variations between differing numbers of flows in a measurement period can be accounted for. These variations in flows are typically daily and weekly variations of traffic flows - both the measured observation points from which the traffic is capture are that of a university environment, and as such will tend to be less busy in the early hours of the morning, and on the weekends. One unaccounted variation is that of the seasonal traffic flows, where traffic rates are known to be different during semester time than they are during the university holiday period. The Auckland traffic traces used are from December, and are captured outside of the standard New Zealand A/B University semester cycle, as opposed to the Waikato traceset, which is captured during June, during the New Zealand university year. It is unknown if this was during a semester break, which also will have a depressive

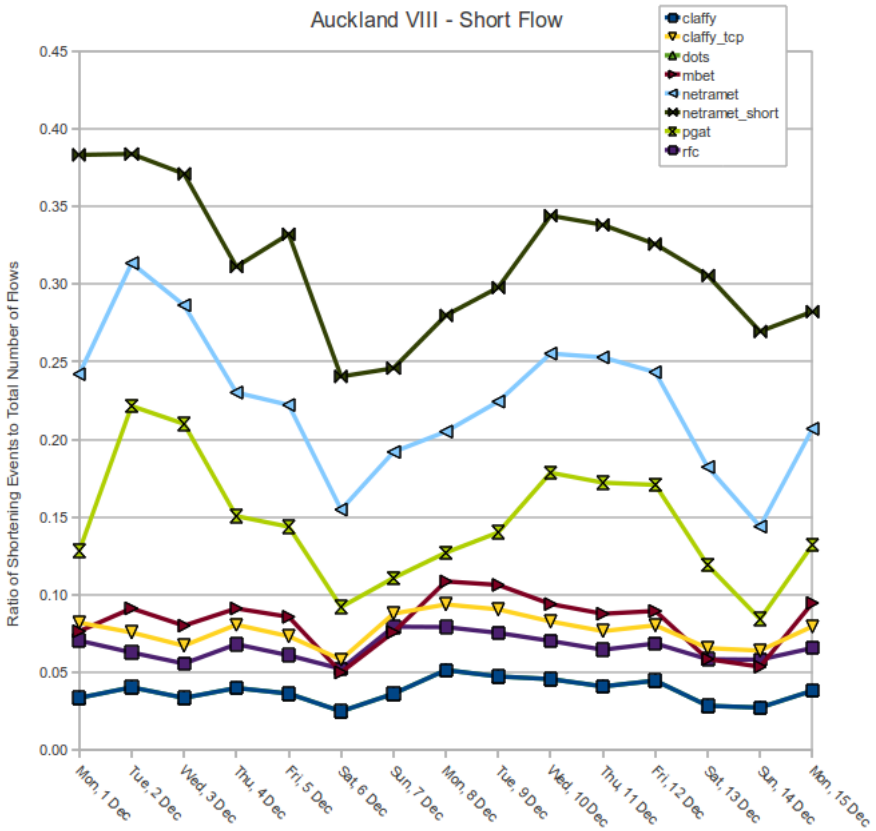

Figure 5.6: Auckland VIII Shortened Flow Incidence

impact on traffic numbers.

Points of interest here are that Claffy, with a 64 second timeout, has the lowest number of flows affected by shortening. Netramet and its derivative have high numbers of shortened flows, due to an aggressive timeout algorithm, which closes flows too early for a firewall, at least in its current configuration. We also note here that Netramet is an algorithm designed for accounting applications. As noted above, in a accounting context, a shortened flow is a side effect of memory efficiency. Whats also interesting here is a weekly cycle, where flows are generally less shortened on the weekend.

Comparing figure 5.7 against figure 5.6 suggests that the incidence of shortening has reduced markedly. The ordering of algorithms affected by shortening has not significantly changed in 4 years, with Netramet and Netramet Short being the most shortened, and Claffy 64 being the least. One possible explanation for this behaviour is that fixed timeout values in firewalls are causing

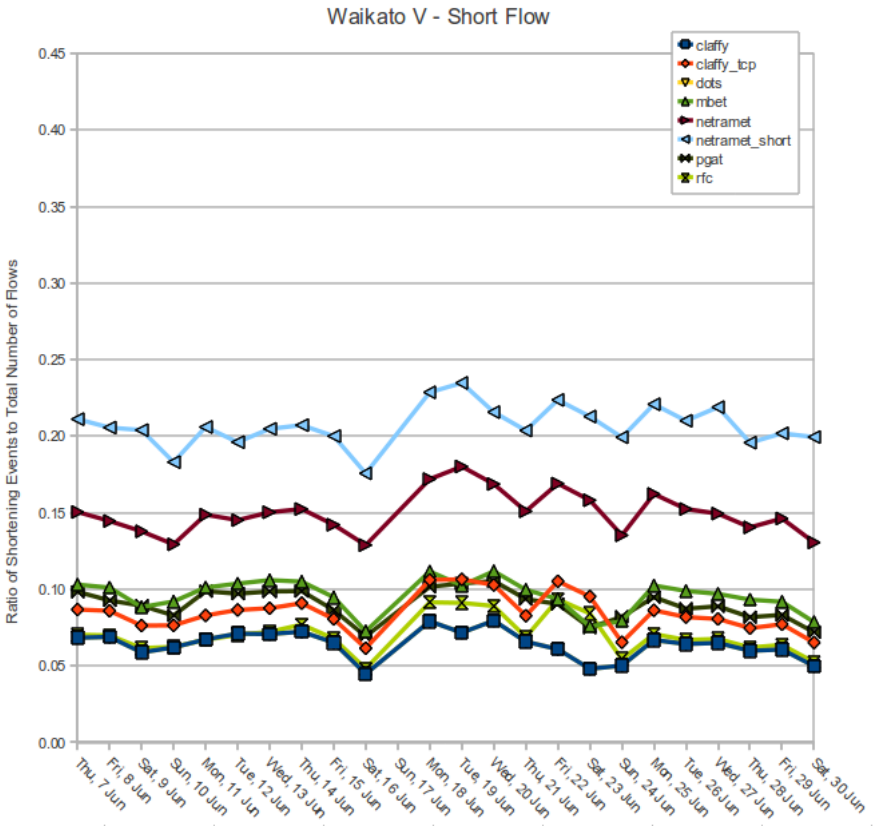

Figure 5.7: Waikato V Shortened Flow Incidence

developers to write timeout aware applications that send keep alive packets to keep the connection open.

Chapter 6

Future Work

Any further work should include a systematic survey of the WITS archive, looking for longitudinal changes in network behaviour. The differences in shortening between the Auckland and Waikato trace sets suggest that this is an area as yet unexplored.

This research did not incorporate IPv6 packets. IPv6 packet handling functionality is present in Libflowmanager, and it should be reasonably easy to add IPv6 capability to this toolset.

As noted in Chapter 4, Section 4.6.2.4, this project encountered issues around modelling TCP state in end hosts. This was primarily due to packet loss between the observer point and an end host. To properly model TCP flow, a detailed state machine that takes account of packet loss needs to be built. This piece of research has not been systematically explored. Further work is required in this area because it would enable the comparison tools to accurately identify lengthening (where a SYN packet is seen in the middle of a established TCP session). Initial indications are that it is not common. Most algorithms will see up to one thousand events in a ten million flow trace, but the current method of measurement is a best-effort approximation.

A further set of measurements that should be undertaken is to check if shortening of TCP flows are actually correctly closed TCP files that re-use the 5-tuple within 120 seconds.

The Cisco Netflow (Chapter 3, Section 3.1) and the Quan and Carpenter (Section 3.10) algorithms were not implemented. This was due to time constraints. Implementing the Netflow algorithm may have real world applications, because it could allow organisations to take a snapshot of traffic with a high speed network capture device, then model what a different firewall configuration could do to their network traffic.

The summariser script step is quite slow, taking around 2 hours to summarise 9 algorithms. This is most likely to be due to an inefficient GZip decompresser - A/B testing on uncompressed files suggests that the Perl GZip decompresser used is about twice as slow as the raw file reader. This is unexpected, since anecdotal evidence suggests that reading compressed files removes large amounts of disk I/O, and should be as fast, if not faster than reading raw files.

The Netramet algorithm has a fixed multiplier of 20. This is not varied at all, and a variation of Netramet could be tested where the multiplier was based in some form on the maximum inter-packet time for a flow, similar to the methodology used by some of the other adaptive algorithms.

Its theorised that a large amount of this runtime for any of the algorithms is Network congestion, rather than CPU utilisation time. This has been strongly inferred by the decrease in time required for an algorithm to run by re-working the algorithms to read and write compressed data. This has not been proven directly by network traffic observation. A peer to peer trace distribution system would be an effective mechanism for the initial start up of a Libflowmanager run of algorithms, given that all the hosts are on a Gigabit network and all read from the front of a network flow.

Chapter 7

Conclusion

This project, which was designed to be part of a larger picture of flow termination research, has successfully delivered on its two main objectives:

1. The gathering together, evaluation and implementation of a a set of previously published flow timeout algorithms

2. The development of a set of comparative tools, with which to run and compare algorithms against each other

The broader value in this work is that it creates a platform for further investigation in this area, with possible avenues of research identified in the Future Work chapter. This work also enables a quantitative method of analyses for flow timeout algorithms, allowing existing algorithms to be compared with new approaches in a systematic way.

Appendix A

MBET Configuration Options

Configuration	T_0 (sec)	S	$\mathcal{P} = \{P_0, \ldots, P_{S-1}\}$ (pkts)
CFG-1	4	5	$\{15, 12, 9, 6, 3, 2\}$
CFG-2	4	5	$\{10, 8, 6, 4, 3, 2\}$
CFG-3	4	5	$\{7, 6, 5, 4, 3, 2\}$
CFG-4	8	4	$\{15, 12, 9, 6, 3\}$
CFG-5	8	4	$\{10, 8, 6, 4, 2\}$
CFG-6	8	4	$\{7, 6, 5, 4, 3\}$

Figure A.1: Auckland VIII High Water Flow Mark for Memory Usage

Appendix B

DoTS timeout Table

MPIT (s)		2	4	8	16	32	64	∞
http	0	0.6364	0.7261	0.7865	0.8414	0.9134	0.9451	1.0000
	10	0.5955	0.6956	0.7642	0.8289	0.9150	0.9495	1.0000
	100	0.4355	0.5644	0.6640	0.7429	0.8871	0.9517	1.0000
	500	0.2959	0.4145	0.5114	0.5911	0.8230	0.9248	1.0000
https	0	0.4089	0.5562	0.6758	0.7549	0.8316	0.9103	1.0000
	10	0.3912	0.5480	0.6770	0.7635	0.8477	0.9320	1.0000
	100	0.1960	0.3577	0.5399	0.6648	0.7895	0.9020	1.0000
	500	0.1850	0.3404	0.5084	0.6540	0.7823	0.8737	1.0000
ftp	0	0.3133	0.3686	0.4349	0.5610	0.6365	0.7065	1.0000
	10	0.2169	0.2820	0.3654	0.5287	0.6252	0.7103	1.0000
	100	0.1074	0.1381	0.2140	0.4368	0.5664	0.6733	1.0000
	500	0.2111	0.2476	0.3301	0.4254	0.5178	0.6159	1.0000
telnet	0	0.3661	0.4172	0.4590	0.5322	0.5995	0.6424	1.0000
	10	0.1124	0.2097	0.2606	0.3961	0.5225	0.6024	1.0000
	100	0.0942	0.1429	0.1930	0.3444	0.5102	0.6201	1.0000
	500	0.1207	0.1835	0.1984	0.2942	0.4532	0.5596	1.0000
smtp	0	0.3168	0.4050	0.5227	0.6055	0.7269	0.8585	1.0000
	10	0.3331	0.4260	0.5632	0.6448	0.7714	0.8486	1.0000
	100	0.1441	0.2161	0.3436	0.4342	0.5813	0.6920	1.0000
	500	0.0601	0.0842	0.1630	0.2320	0.3693	0.4659	1.0000
pop3	0	0.6612	0.8621	0.9337	0.9614	0.9764	0.9889	1.0000
	10	0.6825	0.8570	0.9320	0.9613	0.9771	0.9902	1.0000
	100	0.3325	0.6178	0.7812	0.8694	0.9300	0.9649	1.0000
	500	0.1516	0.3341	0.5075	0.6661	0.7880	0.8638	1.0000
other	0	0.6935	0.7831	0.8277	0.8893	0.9342	0.9594	1.0000
	10	0.4200	0.5331	0.6141	0.7270	0.8554	0.9119	1.0000
	100	0.2334	0.3408	0.4354	0.5548	0.7887	0.8880	1.0000
	500	0.1716	0.2465	0.3306	0.4381	0.7236	0.8532	1.0000

Figure B.1: DoTS flow timeout lookup table

For more information on how the DoTS algorithm interacts with this table, see subsection 3.8 in Chapter 3: Algorithms.

Appendix C

Software

The source to the software used in this dissertation is located on github, and is located at this address: https://github.com/uow-dmurrell/TimeoutSimulator . To compile and install the software:

- Install libtrace3 or greater. On Ubuntu or Debian systems, "sudo apt-get install libtrace3-dev libtrace3", or acquire the software from the Libtrace site at http://research.wand.net.nz/software/libtrace.php
- Install git. Instructions: http://help.github.com/linux-set-up-git/
- Git clone from here: "git://github.com/uow-dmurrell/TimeoutSimulator.git"
- Change into the "TimeoutSimulator/libflowmanager-2.0.0" directory
- Run "autoreconf -i"
- Run "./configure CXXFLAGS=-O0"
- Run "make -j"
- Run "example/example -t"

A memory bug exists in Libflowmanager that occurs when CPPFLAGS -O2 is enabled. As yet, this hasn't been fixed. Using -O0 works around this problem.

This will produce a compiled instance of the Libflowmanager code used in this dissertation. Sample flows, used for debugging algorithms, are located in the "sampleflows" folder.

The example program is the main program that drives this environment, and has fairly detailed help, accessible using the "-h" switch.

For example, to run one of the sample trace files, run "example/example pcapfile:../sampleflows/output3.pcap", and a csv file with the trace output should be listed in the current working directory. The example program can be run from anywhere, as long as a path to the executable is specified.

For debugging, the actual executable is located in "example/.libs/example", - the file listed in the previous paragraph is a libtool script, which locates the program file libraries for the program correctly.

Tools to run comparisons on the cluster are located in the tools folder. Some of this is hardcoded with the authors username (dtm7), but this is a fairly trivial fix. The file that will kick off a tracefile or series of tracefiles is called "ftptest.pl". This is the script mentioned at the top of Figure 4.1 in Chapter 4. This "ftptest.pl" script calls "sorter.pl" to sort, and gzip the libflowmanager output, and then schedules a the summariser script, called "readline.pl". The file "readline.txt" in the tools directory has some further notes on the data structure used in the summariser script.

Libflowmanager is licensed under the GPL version 2. Any scripts or code located uploaded by the author to the github TimeoutSimulator repository also fall under the GPL version 2.

References

Shane Alcock. Libflowmanager. http://research.wand.net.nz/software/libflowmanager.php, 2011.

F. Audet and C. Jennings. *Network Address Translation (NAT) Behavioral Requirements for Unicast UDP*. Internet Engineering Task Force, January 2007. URL http://www.ietf.org/rfc/rfc4787.txt.

R. Braden. *RFC 1122 Requirements for Internet Hosts - Communication Layers*. Internet Engineering Task Force, October 1989. URL http://tools.ietf.org/html/rfc1122.

Mills C. Ruth G. Brownlee, N. *Traffic Flow Measurement: Architecture*, October 1999. URL http://tools.ietf.org/html/rfc2722.

N. Brownlee. Some Observations of Internet Stream Lifetimes. In *Passive and Active Network Measurement Workshop (PAM)*, pages 265–277, Boston, MA, Mar 2005. PAM 2005.

N. Brownlee and M. Murray. Streams, Flows and Torrents. In *Passive and Active Network Measurement Workshop (PAM)*, Amsterdam, Netherlands, Apr 2001. RIPE NCC.

Cisco. Netflow services solutions guide, 2007. URL http://www.cisco.com/en/US/products/sw/netmgtsw/ps1964/products_implementation_design_guide09186a00800d6a11.html.

K. Claffy, H. Braun, and G. Polyzos. A parameterizable methodology for

Internet traffic flow profiling. *IEEE Journal on Selected Areas in Communications*, (8):1481–94, Mar 1995.

cplusplus.com. C++ Reference: STL Containers: Multiset. http://www.cplusplus.com/reference/stl/multiset/, 2011a.

cplusplus.com. C++ Reference: STL Containers: List: Size. http://www.cplusplus.com/reference/stl/list/size/, 2011b.

S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. *NAT Behavioral Requirements for TCP*. Internet Engineering Task Force, October 2008. URL http://www.ietf.org/rfc/rfc5382.txt.

Paul Hsieh. "Superfasthash". http://www.azillionmonkeys.com/qed/hash.html, 2008.

Intel. VTune Amplifier XE 2011 for Linux. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/#resources, 2011.

P. Kohler. Netflow for accounting analysis and attack. In *Networkers 04*, 2004. URL http://www.cisco.com/networkers/nw04/presos/docs/NMS-2032.pdf.

Paul Marquess. IO::Uncompress::Gunzip: Read RFC 1952 files/buffers. http://perldoc.perl.org/IO/Uncompress/Gunzip.html, 2011.

Zhou Ming-zhong, Gong Jian, and Ding Wei. Study of dynamic timeout strategy based on flow rate metrics in high-speed networks. In *Proceedings of the 1st international conference on Scalable information systems*, InfoScale '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-428-6. doi: http://doi.acm.org/10.1145/1146847.1146852. URL http://doi.acm.org/10.1145/1146847.1146852.

L Qian and B. Carpenter. Some observations on individual TCP flows behaviour in network traffic traces. Draft, not yet published, 2011.

Bo Ryu, David Cheney, and Hans werner Braun. Internet flow characterization: Adaptive timeout strategy and statistical modeling. In *in Proc. Passive and Active Measurement workshop*, page 45, 2001.

Gurusamy Sarathy. Data::Dumper: Stringified perl data structures. `http://perldoc.perl.org/Data/Dumper.html`, 2010.

WAND. Waikato internet traffic storage. `http://wand.net.nz/wits/catalogue.php`, 2011.

University of Waikato WAND Group. The dag project. `http://dag.cs.waikato.ac.nz/`, 2001.

Junfeng Wang, Lei Li, Fuchun Sun, and Mingtian Zhou. A probability-guaranteed adaptive timeout algorithm for high-speed network flow detection. *Comput. Netw.*, 48:215–233, June 2005. ISSN 1389-1286. doi: 10.1016/j.comnet.2004.11.005. URL `http://dl.acm.org/citation.cfm?id=1648529.1648724`.