ANGEL Flow Meter
Software Architecture Design Document

Jason But
Centre for Advanced Internet Architectures, Technical Report 070228C
Swinburne University of Technology
Melbourne, Australia
jbut@swin.edu.au

***Abstract*—The Automated Network Games Enhancement Layer (ANGEL) project aims to leverage Machine Learning (ML) techniques to automate the classification and isolation of interactive (e.g. games, voice over IP) and non-interactive (e.g. web) traffic. This information is then used to dynamically reconfigure the network to improve the Quality of Service provided to the current interactive traffic flows and subsequently deliver improved performance to the end users. Within this scope, the project will develop protocols that allow the adjustment of Consumer Premise Equipment (CPE - eg. cable/ADSL) configuration to provide better quality of service to interactive flows detected in real-time.**

This document describes the basic design motivation of the Flow Meter Software Component of ANGEL. The Flow Meter is responsible for capturing packets off a network connection, collating the statistical properties and forwarding this information to the Flow Classifier Component.

I. Introduction

This document details the software design decisions made when building the ANGEL Flow Meter. In particular we discuss how the design takes scalability into account as well as attempting to overcome any limitations based on the underlying packet capture facility.

II. Flow Meter Requirements

We start by repeating the minimum requirements for the Flow Meter as stated in the ANGEL Architecture document [1]. These provide a useful reference when considering any specific constraints in the software architecture.

An ANGEL System can contain numerous ANGEL Flow Meters. The number and configuration of each Flow Meter is dependent on the network design and configuration, the number of current or potential ANGEL users, and the amount of network traffic flowing through the network at metering locations.

The primary tasks of a Flow Meter are to:

- Monitor a copy of network traffic at a particular location
- Filter the traffic into individual network flows
- Forward captured packet and flow information to the ANGEL Flow Classifier

Key design restrictions for all aspects of the ANGEL Flow Meter are:

- Two or more network connections, one to communicate with the Flow Classifier and ANGEL Database, the other(s) to receive a copy of all network traffic from the tap(s)
- Configured with (either locally or from the ANGEL Database):
  - IP Address of the ANGEL Database
  - IP Address of the ANGEL Flow Classifier to communicate the packet statistics to
  - IP network connection details of ANGEL Customers - allows filtering of captured packets
  - Flow timeout values
- Optionally filter captured packets to only consider currently registered ANGEL users - packets that match the active list of IP addresses to be monitored are kept while other packets are immediately discarded. This is particularly useful in situtations where a small proportion of users are ANGEL enabled, as fewer Flow Meters need be deployed. In situations where a large proportion of users are ANGEL enabled, it may be simpler to process all traffic including non-ANGEL enabled users.
- Packet statistics are always communicated to a single Flow Classifier
- Packet statistics for multiple flows can be sent in a

From February 2007 and July 2010 this report was a confidential deliverable to the Smart Internet Technologies CRC. With permission it has now been released to the wider community.

single packet to the Flow Classifier
- If a currently active flow becomes inactive for the configured timeout period, resources used to hold the flow state will be released and the Flow Meter will signal the Flow Classifier that the flow is terminated.

A. Key Restrictions

The ANGEL Flow Meter will use **libpcap** [2] as the underlying packet capture facility. This means that any limitations of the PCAP library must be taken into account:

- Packets are timestamped by the kernal and/or PCAP. The accuracy of packet timestamps are therefore dependent on the underlying hardware and OS implementation
- PCAP expects the application to do all processing on the captured packet before supplying any other packets - subsequent packets are buffered. Memory resources for captured packets are freed as soon as the application tells PCAP it has completed processing
- The PCAP buffer allows packets to be queued if they arrive in quick succession while the previously captured packet is processed. The PCAP buffer size is configurable, a larger buffer can be used to ensure that packets are not lost due to buffer overflow - on BSD we should use a a buffer size of 512kB rather than the default 32kB. Even so, it is important that the mean packet processing rate must be greater than the mean packet arrival rate
- While the PCAP buffer size is configurable, there is a limit at which it will overflow and captured packets will be lost unless processing time can proceed at a rate greater than the packet arrival rate
- The Flow Meter should be built on FreeBSD (or other BSD variant) as the kernel packet capture facility on Linux based systems has a tendency to re-order packets prior to delivery to the capture application

Speed of access to the external database is undetermined since it may be located on a physically seperate machine. It is imperative that accesses to the database are minimised and configured in such a way so as to not limit the processing capability of the Flow Meter.

The Flow Meter must send information to the Flow Classifier for each captured packet. To minimise the amount of data that the Flow Classifier is responsible for handling it is imperative that the amount of data generated by the Flow Meters is minimised

We propose to implement redundancy via the use of two or more machines running the same software. The machines will use the CARP protocol [3] to elect a master out of the set of machines. The master Flow Meter will communicate data to the Flow Classifier, when it is detected that the master is offline, one of the slave machines will be elected master to assume these duties. We expect a typical system would use two machines acting as a single Flow Meter unit.

B. System Configuration

The Flow Meter must run using some configuration options. The ANGEL Architecture allows for configurable options to be stored in the database. Even so we must also allow for configuration options to be stored locally on the Flow Meter. Further, some configuration options must be known prior to using the database.

1) Local Configuration Information: The Flow Meter should use a local configuration file which stores:

- Capture Interface - Network interface on which to capture packets
- Database Contact Information - Details required to contact and use the external database. This could include details such as IP address, username, password, etc.
- Capture Size (optional) - Number of bytes to capture from each packet, if not specified use a default value. Can also be specified in the database
- Flow Meter Contact Information (optional) - Details of the Flow Classifier to forward data onto could be stored here. Any values in the local configuration file are overidden by similar data in the database
- Flow Timeout (optional) - Details of how long to wait to timeout flows. Any values in the local configuration file are overidden by similar data in the database

Any optional configuration values (excluding Capture Size) **must** be stored in either the local configuration file **OR** the database. These values are required for the Flow Meter to run.

C. CARP Configuration

The Flow Meter is designed to provide failover redundancy through the use of CARP (Common Address Redundancy Protocol [3]) - a tool which has been created and maintained by the OpenBSP project. CARP is a free alternative to the VRRP (Virtual Router Redundancy Protocol) and the HSRP (Hot Standby Router Protocol). The tool enables multiple Flow Meter units to share a single, virtual network interface between them, so that if

any machine fails, another can respond instead. The Flow Meter is coded to keep polling the machine for master state. The Flow Meter functions correctly without CARP except that no failover redundancy features are present.

CARP must be installed and configured in the underlying OS. At configurable intervals (1 second by default - set by the `advbase` parameter), the master advertises its operation on IP protocol number 112. If the master goes offline (the backup CARP host doesn't see an advertisement from the master for 3 consecutive advertisement intervals), the backup system in the CARP group begins to advertise.

The host that is able to advertise most frequently becomes the new master. The CARP advertisement skew (*advskew*) parameter is used to skew the advertisement interval of a less preferred host in becoming master. We recommend setting the Flow Meter unit that is expected to be the master with a `advskew` value of 0, and the backup Flow Meter with `advskew` set to 100. The total advertisment interval is calculated as `advbase + (advskew / 255)`.

The steps required to configure CARP on a FreeBSD OS designed to function as a Flow meter involves:

1) Re-compile the kernel with the following options:

```
device carp
```

2) Configuration of the Flow Meter Master:

```
ifconfig carp0 create
ifconfig carp0 vhid 1 pass angel 1.1.10.1/24
ifconfig carp1 create
ifconfig carp1 vhid 2 pass angel 1.1.20.1/24

sysctl net.inet.carp.preempt=1
```

3) Configuration of the Flow Meter Backup:

```
ifconfig carp0 create
ifconfig carp0 vhid 1 advskew 100 pass angel 1.1.10.1/24
ifconfig carp1 create
ifconfig carp1 vhid 2 advskew 100 pass angel 1.1.20.1/24

sysctl net.inet.carp.preempt=1
```

III. Multi-Threaded Design

The Flow Meter will be designed as a multi-threaded application. This structure allows optimum scalability with the ability to run multiple processes simultaneously while the meter is blocked waiting for the OS, while also allowing for improved execution speed since the processes can communicate with each other via shared memory access.

In this section we will highlight the main threads of execution and their primary tasks. Subsequent sections will describe the processing of each thread in more detail.

A. Packet Capture

One of our key restrictions is the need to process packets returned by the PCAP library as quickly as possible such that the number of packets dropped is minimised. However, the PCAP library also releases memory used to store captured packets as soon as the capture application returns from handling that packet.

Our goal is to optimise the packet capture facility by allocating a thread of execution that is primarily responsible for capturing a packet, making a copy, and then queueing it for processing by another thread (see Figure X). The concern is that the size of the PCAP buffer is static. However, by using a generic queue structure within the Flow Meter application, we are essentially moving buffering of captured packets to a dynamic queue limited in size by available memory.

With the capture thread only performing these basic tasks, processing time for an individual packet is limited and coupled with an appropriate PCAP buffer size we should have a scenario where no captured packets are lost given that the mean packet processing rate is greater than the mean packet arrival rate.

It is necessary to make a copy of the packet prior to queueing it to allow PCAP to release memory resources without losing the captured data.

B. Packet Processing

This thread is responsible for pulling packets captured by the Packet Capture Thread out of the inter-thread queue, and calculating packet statistics to deliver to the Flow Classifier. This information is then queued for another thread to manage delivery to the Classifier. The primary purpose of dividing these tasks is that communication of data to the classifier is periodic and can result in blocking of the thread. This design allows for captured packets to continue to be processed even if this is the case.

For each packet that is captured, this thread must calculate a Flow Hash[1]. If the generated hash is new - this is the first packet of this flow - then the mapping between the hash and the flow tuple (source/destination) information must be stored in the database. In all cases, the flow hash, packet size and packet timestamp are queued for the network thread to deliver to the Flow Classifier.

It should be noted that only the Master Flow Meter is allowed to update the database.

[1] 32 bit value based on the flow ID tuple information <sourceIP; sourcePort; destIP; destPort; Protocol>

C. Network Thread

The network thread is responsible for de-queueing summarised packet statistics generated by the Packet Processing Thread and sending them to the Flow Classifier. It would be in-efficient to send a single packet to the Flow Classifier for every captured packet, and the protocol specifications allow for statistics of multiple packets to be encapsulated within a single packet for delivery to the Flow Classifier.

However, waiting for a certain number of packets prior to delivery to the Classifier increases the time between traffic generation and classification. The thread is designed to wait until one of two events occur:

1) Enough packets are captured/processed to fill[2] a data packet to the Classifier
2) A timeout has triggered since the last data packet delivered to the Classifier

After one of these events occur, the currently (potentially paritially filled) generated data packet is forwarded to the classifier. It should be noted that only the Master Flow Meter is allowed to transmit to the Flow Classifier.

D. Primary Thread

As well as launching the other threads to perform the basic work of the Flow Meter, the Primary Thread has a handful of other tasks it is required to perform:

- Periodically poll the database for updated configuration information. This includes configuration settings for both flow timeout values and contact details for the Flow Classifier. The polling period is not required to be frequent as this information is likely to remain stable for extended periods of time and reading these values at time intervals measured in minutes would be acceptable
- Garbage collect information on current flows and remove their details from the database. This repetitive process must scan through details of each flow and - if there has been no activity for the specified timeout period - remove the corresponding flow hash entries from the database. Again, the frequency of this task does not have strict time requirements. Indeed, performing this task too often would come at the expense of the Packet Processing Thread being more likely to have to wait to obtain access to the database as required. The primary cost of infrequent garbage collection is that flow information lives for longer periods of time. For each flow that has been determined to be terminated, the Primary Thread must create and queue the *End-of-flow* message for the terminated flow to the Network Thread for delivery to the Flow Classifier

[2]The data packet is not larger than the network MTU

E. Inter-Thread Communication

All threads need to be able communicate data amongst each other. In threaded applications, this is typically done through the use of shared memory and variables. Problems occur when two threads need to access the same memory block at the same time. This situation is typically solved via the use of Mutual Exclusions and Semaphores which cause some threads to block while protected code is being executed.

Common difficulties with this approach arise because:

- To ensure that we have thread-safe code we need to wrap as much as possible inside a protected space. This ensures that we don't *miss* a shared variable modification which could cause the code to misbehave
- To develop an optimal/efficient system, we need to wrap as little as possible inside a protected space. This ensures that code is only blocked from executing when it is absolutely necessary, maximising CPU usage

All access when pushing data onto and popping data off inter-thread communication queues **must** be protected with a mutual exclusion. If a thread wants to push data onto a queue it **must** wait until the other thread has finished retrieving from the queue.

All access to the database **must** be protected with a mutual exclusion. Similarly all access to the internal list of currently active flows **must** be protected with a mutual exclusion. This is important as one thread isused to build and update the list while another is performing garbage collection duties on the list.

F. Development Tools

The Flow Meter makes use of the ACE library/toolkit [4]. We use this toolkit to provide C++ wrappers around common network functionality (sockets) as well as to provide tools to aid thread creation, thread management and inter-thread communication.

G. Thread Implementation

The Flow Meter is to be implemented using a C++ Class Oriented Structure. A class is to be defined that should create, launch and manage the thread. Each of these classes may use further helper classes to assist in the implementation.

IV. Capture Thread

This thread should be implemented in a class called SniffTask. The SniffTask class should:

- Be inherited from the ACE ACE_Task class. This class implements thread safe queues to allow other threads to post messages to this thread
- Be created with a pointer to the Packet Processing Thread so that messages can be posted to the Packet Processing Thread Queue
- Obtain the Capture Interface and Capture Size details from the Global Configuration database

The class should open and initialise packet capture using the PCAP library functions. Once running the service thread should continuously:

- Get the next packet from the PCAP library
- Make a copy of the captured packet details including type, timestamp and captured payload
- Push the copied information onto the message queue for the Packet Processing Thread
- If signalled to terminate by the primary thread we need to stop capturing packets and terminate the thread

A. Singleton Class

There can only be one instance of SniffTask. The class should be implemented as a Singleton such that if a second instance is created it fails with an appropriate error message. This allows checking for this error in parts of the code that create the Capture Thread.

B. Construction/Initialisation

The class constructor should open the specified interface for live capture, ready to be used by the thread. In this way when the thread is actually launched we are able to begin capture immediately and any potential configuration errors are caught at construction phase. The capture device should be closed in the close() method which is automatically called when the main thread method terminates.

C. Main Processing

Processing is performed in the **svc()** method. How this is implemented is primarily dependent on how the PCAP library works. We call **pcap_loop()** to begin capture. Capture continues - and this function call will not return - until capture is stopped or suspended. As such the code within the **svc()** method should check the message queue to see if any messages (particularly the thread shutdown message) exist and act on these messages. Once the queue is empty we call **pcap_loop()** to begin capture and processing of packets.

As packets are captured, a registered callback function is called to process each packet. Within this function we should check to see if the thread message queue is empty. If not empty we need to suspend capture so that the original call to **pcap_loop()** is broken and **svc()** can handle the message. Either way the callback should make a copy of the captured data and post it to the Packet Processing Thread class.

V. Packet Processing Thread

This thread should be implemented in a class called ProcessTask. The ProcessTask class should:

- Be inherited from the ACE ACE_Task class. This class implements thread safe queues to allow other threads to post messages to this thread
- Be created with a pointer to the Network Thread so that messages can be posted to the Network Thread Queue

Once running the service thread should continuously:

- Get details of the next packet from the input queue - posted by the Capture Thread
- Parse the packet to extract flow tuple information, including source and destination addresses and Port Numbers, as well as Transport Layer Protocol information
- Generate a 32 bit hash of the flow tuple identification
- If the packet starts a new flow, add the hash and tuple information to the database and local flow map
- Queue the hash, timestamp, packet size and flow ID flags to the Network Thread for transmission to the Flow Classifier

A. Main Processing

Processing is performed in the **svc()** method. This should run as an infinite loop which dequeus packet information and processes it prior to passing it to the Network Thread. The processing tasks include separating packets into individual flows, calculating an identifying hash for each flow and combining this information for the network thread.

ProcessTask uses a series of helper classes to achieve its task. In particular a set of multi-layered classes based on the Packet Class are used to parse the captured packet and to enable extraction of flow ID tuple information. This set of classes should be expandable to include new intermediate layers to extend the range of devices that the Flow Meter can run on and the number of protocols

that can be processed. Each sub-class should parse the header of a particular protocol before deciding which (other) sub-class should parse the subsequent layer. More information on the Packet sub-classes is provided in a later section.

An internal map of current flows must be kept as it is impractical to continuously probe the (possibly external) database for each packet we see. This map should keep track of the hash for each flow as well as the timestamp of the last witnessed packet for the flow. This data allows for garbage collection of flows that have no activity within a certain timeout period. The map should also be searchable to detect creation of a new flow such that the corresponding identification information can be written to the database.

The flow map must also be accessed by the primary thread for garbage collection purposes. As such access to maintaining the map must be thread-safe. To keep code centralised, all flow map updating, as well as the map itself, should be implemented as a separate class. This class - FlowMap - will be described in a later section.

B. The Hash

The primary purpose of the hash is to minimise the amount of information being passed between the Flow Meter, Flow Classifier and Client Manager. The Client Manager can extract IP addresses from the database when it needs to for a given flow hash. As such the hash - nominally a 32 bit value - must uniquely identify all flows. Given that:

- Flows have a limited lifetime
- A Flow Meter is located to service a sub-set of clients of an ISP - the source or destination IP addresses must be an IP address belonging to the ISP
- The number of concurrent flows generated by a subset of customers is limited

We would expect that 32 bits should be a large enough value to uniquely identify all current flows. However, it is imperative that the hash is selected to minimise the number of collisions given the constraints of possible input values. This document will not comment on the actual hash algorithm other than to state that:

- The input to the hash function must include all data to uniquely identify a flow:
  - Source/destination IP addresses - either IPv4 or IPv6
  - Source/destination Port Numbers - or zero if just an IP packet
  - Transport Layer Protocol - to differentiate TCP and UDP flows using the same end-point identifiers
- The hash function should ensure that the same hash is generated for packets of the same flow (ie. regardless of the source/destination direction of the packet). The design approach used here will be to sort the IP addresses such that the *lowest*[3] IP address is always considered to be first when generating the hash. If IP address orders are swapped, then the ports should also be swapped

VI. Network Thread

This thread should be implemented in a class called NetworkTask. The NetworkTask class should:

- Be inherited from the ACE ACE_Task class. This class implements thread safe queues to allow other threads to post messages to this thread
- Obtain the local port number to bind the UDP socket to from the Global Configuration database.

Once running the service thread should continuously:

- Get details of the next packet from the input queue - posted by the Process Thread
- Append the data to a buffer
- If the buffer has exceeded a maximum size **OR** a timeout trigger since the last transmission has fired, then send the buffer to the Flow Classifier

A. Construction/Initialisation

The class constructor should create the UDP socket for later communications by the thread. The socket should be closed in the destructor.

B. Main Processing

Processing is performed in the **svc()** method. This should run as an infinite loop which dequeues packet information and queues it for transmission to the Flow Classifier. Packets should be queued until either:

- The number of buffered packets exceeds a certain value
- A timeout has triggered since the last data packet was sent to the Flow Classifier

At this stage, the current queued buffer, regardless of size, should be delivered to the Flow Classifier using the UDP socket.

[3]The definition of *lowest* is up to the implementor but must be consistent

VII. Packet Parsing

Packet parsing is a layered process, the lower layer packet header must be parsed before any higher layered data can be processed. Given that all layers use these same basic steps, a class based approach works well where all packet parsers are implemented as classes inherited from a common base class Packet.

The Packet class defines what values a parsed packet should return, for our case this will include the source and destination IP address/port numbers, as well as the Layer 4 protocol information. Since the packets are layered, the header information is stored as instances of Packet classes within other instances. Where a particular layer does not know how to return certain information (eg. The Ethernet layer does not know the port numbers), it returns the result from the recursive request on the contained packet (in this case the IP layer which would subsequently call the TCP layer). The base class can implement default return values that call encapsulated packets or return NULL values to signify that the requested information is not available for this packet.

A. Parsing

All parsing is done by the Packet constructor. At the top level we know what format the packet was captured in because this is returned by PCAP. We can use this information to construct an instance of the appropriate packet parser for that protocol.

Each class constructor should parse the header and extract appropriate bits of information for later use. It should also examine the packet header to determine the protocol of the encapsulated packet and then call the appropriate constructor to further parse the packet. This allows new protocols at multiple layers to be inserted into the parsing chain:

- The initial implementation will parse Ethernet, encapsulated IPv4 and IPv6 and encapsulated TCP and UDP
- A single TCP parser is used for both IPv4 and IPv6 packets
- A single UDP parser is used for both IPv4 and IPv6 packets
- A new layer (say ATM) could use existing IPv4, IPv6, TCP and UDP parsers to further parse the packet once a new ATM level parser is constructed

The implementation in the ANGEL Flow Meter is based in spirit on the packet parser used in NetSniff [5]. NetSniff was developed by the Centre for Advanced Internet Architectures and the license to use this portion of NetSniff has been granted by the author to use in the ANGEL project.

B. IP Addresses

To simplify higher layer code, we would like to use a single class to represent both IPv4 and IPv6 addresses. The implementation used by the Flow Meter is based on that released in the socketcc library - a C++ Socket/Thread library [6]. socketcc was written and developed by one of the ANGEL developers and has been released to the public domain.

VIII. Flow Map

The Flow Map should use a C++ STL map to map flow hashes (32 bit values) to timestamps. It is also necessary to store the most recent timestamp to allow for garbage collection timeouts. Both these variables should be maintained in a class called FlowMap.

A. Singleton Class

There can only be one instance of FlowMap. The class should be implemented as a Singleton such that if a second instance is created it fails with an appropriate error message. Further, the class should not actually be creatable. Rather a call to the static member method GetInstance() should return a pointer to the singleton instance, or create an instance and return the pointer if current instance exists. Once this pointer is returned methods can be called on the instance.

B. Updating the Map

The ProcessTask thread must be able to update the map with new flow information. This involves:

- Checking to see if an entry for this flow already exists in the Map
- Storing the timestamp in the map for the given hash
- Updating the last seen timestamp member variable
- Returning to the caller if the packet is for a new flow

C. Garbage Collection

Periodically called by the Primary Thread to clean entries from the Map. We need to loop through each entry in the map, comparing the current time with the last-seen timestamp and using the timeout threshold to select which entries to cull. These entries must be placed in a temporary list.

We then need to loop through the *cull*-list and actually delete the entries from the map - a map cannot be modified while we are iterating through it, hence the two stage process.

The *cull*-list must now be used to formulate the *End-of-flow* information to send to the Flow Classifier. An appropriate data block must be constructed for each hash in the map and queued to the Network Task for delivery to the Flow Classifier on the next scheduled transmission.

D. Making it Thread-Safe

The FlowMap class should also contain a mutual exclusion that is used within the method implementations to lock access to the contained STL Map and other shared member variables.

1) Updating a Flow: A Flow Hash and timestamp is provided. The map is updated with the information and **TRUE** is returned if the hash signifies a new flow. The search through the STL map and the updating/addition of the hash to the map must be protected with the mutual exclusion.

2) Removing a Flow: The specified hash must be removed from the STL Map, which must be protected with the mutual exclusion.

3) Removing timed out flows: The mutual exclusion need only exist when we are modifying the STL map (i.e. during the second of the three stages of the garbage collection process). It is not required to lock the mutual exclusion when creating the *cull*-list nor when iterating through this list to create *End-of-flow* messages to deliver to the Classifier.

This approach ensures that the Process Task is able to acquire the Mutual Exclusion to check and update flows in the list while Garbage Collection is occuring

ACKNOWLEDGMENT

This work was supported from 2005 to early 2007 by the Smart Internet Technology Cooperative Research Centre, http://www.smartinternet.com.au.

REFERENCES

[1] J. But *et al.*, "ANGEL Architecture Document," CAIA, Tech. Rep. 070228A, February 2006, http://caia.swin.edu.au/reports/050204A/CAIA-TR-050204A.pdf.

[2] Lawrence Berkeley National Laboratory, "TCPDUMP/LIBPCAP Public Repository," January 2006, http://www.tcpdump.org/.

[3] OpenBSD Community, "PF: Firewall Redundancy with CARP and pfsync," January 2007, http://www.openbsd.org/faq/pf/carp.html.

[4] D. Schmidt, "The adaptive communications environment," January 2007, http://www.cs.wustl.edu/~schmidt/ACE.html.

[5] Centre for Advanced Internet Architectures, "NetSniff," January 2007, http://caia.swin.edu.au/ice/tools/netsniff.

[6] J. But, "SocketCC," January 2007, http://sourceforge.net/projects/socketcc.