ANGEL Flow Classifier Software Architecture Design Document

Jason But
Centre for Advanced Internet Architectures, Technical Report 070228D
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 Classifier Software Component of ANGEL. The Flow Classifier is responsible for analysing packet/flow statistics compiled by the system Flow Meter(s) to classify the flows in realtime, and then to forward any changes to Classification in any current flows to the Client Manager Component.

I. Introduction

This document details the software design decisions made when building the ANGEL Flow Classifier. In particular we discuss how the design takes scalability into account as well as looking at the modularity of a system that allows for replacement of the module that actually performs the classification. Also of note is how the software is designed to meet the requirements of the Flow Classifier within the scope of the ANGEL Project.

II. Flow Classifier Requirements

We start by repeating the minimum requirements and tasks from the ANGEL Architecture document [1]. These provide a useful reference when considering any specific constraints in the software architecture.

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.

An ANGEL System consists of a single Flow Classifier. System scalability **SHOULD** be implemented either as a multiple parallel ANGEL systems or through the use of load balancing within a cluster of machines.

The primary tasks of a Flow Classifier are:

- Classify flows based on the packet statistics provided by any Flow Meters within the system
- If a new flow is classified as requiring prioritisation signal this information to the Client Manager
- If an existing flow changes its classification, signal this information to the Client Manager
- Purge stored flow information when that flow terminates
- When a flow terminates signal this information to the Client Manager

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

- Configured with (either locally or from the ANGEL Database):
  - IP Address of the ANGEL Database
  - IP Address of the ANGEL Client Manager to communicate the flow classification information to
  - Flow timeout values
- Store state information for currently active flows such that a change in state can be detected
- If any of the following classifications are made, signal the Client Manager with the flow hash of the flow
  - A new flow is categorised as realtime
  - A previously non-realtime flow is categorised as realtime
  - A previously realtime flow is categorised as non-realtime
  - A flow is determined to have terminated
- Flow classifications are always communicated to a

single Client Manager
- Packet statistics for multiple flows can be sent in a single packet to the Flow Classifier

A. Key Restrictions

The ANGEL Flow Classifier should be built as a system that uses both a forked process AND threaded implementation. The use of multiple underlying processes allows the Classifier to take advantage of Clustered OS implementations such as OpenMosix [2] that implement load balancing of multiple processes. This task cannot be achieved using threads since threads take advantage of shared memory which is either not possible or not efficient with tasks spread across multiple machines.

While processes should be used to implement support for load-balanced systems, it is viable to use threads within these processes where this may facilitate implementation.

The external database is used by the Flow Classifier to retrieve configuration information. 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 Classifier.

The Flow Classifier must send information to the Client Manager for each change in flow classification state. It is anticipated that the amount of data generated here will be minimal:

- A good classifier will not repeatedly change its classification for a flow, it is expected that each flow will generate at most two or three classifications in its lifetime
- Notification (to the Client Manager) is not required if the flow is initially classified as non-realtime. Since the majority of flows fall into this category it is expected that the actual number of notifications will be minimised

We propose for redundancy to be handled by clustering techniques. The prototype Flow Classifier will be implemented on the Linux based OpenMosix clustered system which implements load balancing by launching processes on different machines in the cluster. The OS will automatically use whatever machines are available in the cluster - including machines that newly come online. The scenario of a machine failing can be detected - the flows that were currently being classified on those machines will obviously not be classified however the Classifier should be designed such that future packets within those flows will be allocated to new processes on other machines for future classifications - resulting in minimal extra time to classification.

B. System Configuration

The Flow Classifier 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 Classifier. Further, some configuration options must be known prior to using the database.

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

- Database Contact Information - Details required to contact and use the external database. This could include details such as IP address, username, password, etc.

Other configuration information **must** be stored in the database. This includes:

- Client Manager Contact Information - Details to forward classification information to.

III. Multi-Process/Threaded Design

The Flow Classifier will be designed as both a multi-processed and multi-threaded application. This structure allows scalability and redundancy through the use of processes with OpenMosix and threads within these processes to improve inter-thread communication and usage of CPU resources on an individual Classifier Cluster machine.

The processes will use standard inter-process communication techniques to forward information between them, threads running within a single process will use standard memory sharing techniques (eg. Mutual Exclusions) to share information and to protect shared data.

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

The Flow Classifier will not have information about the source and destination IP address/Port Number tuples identifying a flow. Within the Flow Classifier all flows are identified by a 32-bit Flow Hash which is calculated by the Flow Meter from the flow tuple data. This is done to minimise the data being sent by the Flow Meter to the Flow Classifier over the network. The Classifier can use the Flow Hash field to sort packets into their independent flows and perform classification. The Flow Hash must be used when communicating classification output to the Client Manager - the Client Manager will use this

data to extract tuple information from the database to determine which CPE device(s) need to be informed about independent flow classification decisions.

A. Parent Process

The parent process is the primary process of the Flow Classifier that receives communications from the assorted ANGEL Flow Meters and distributes classification change notifications to the ANGEL Client Manager. Other responsibilities of the parent process include managing Child Processes to perform the classification tasks.

Essentially we should consider this process to play the co-ordinating role for the Classifier implementation. In order to achieve this task, we have designed the Parent Process to be implemented as three concurrent threads working together to perform the actions required. These threads are detailed below.

1) Meter Network Thread: The Meter Network Thread is responsible for receiving captured packet information from the systemwide Flow Meters and forwarding them to the appropriate Child Processes for classification. This task involves the management, creation and removal of Child Processes as required to support the processing load, as well as maintaining the internal database of which Child Processes are classifying which individual network flows.

System configuration information will be used to determine the number of individual flows to be assigned to each Child Process, the Meter Network Thread must:

- Keep track of individual flows allocated to processes
- Create new Child Processes for new flows where no existing Child Processes have spare capacity
- Destroy Child Processes as flows terminate and the number of flows being handled by the Child Process drops to zero

The packet statistics for flows must be passed to the appropriate Child Process for handling and classification. Since this task is executed in a seperate process, this must be managed by a standard inter-process communication technique. In order to facilitate this a series of communication channels must be established between the Parent Process and each individual Child Process. The application structure is not a true peer-to-peer environment, Child Processes do not need to communicate with - nor be aware of - other Child Processes. There is no requirement for communication channels to exist between Child Processes - only the direct tree-like structure between the Parent and each Child.

2) Manager Network Thread: The Manager Network Thread is responsible for receiving classification information from the Child Processes (after a flow classification has changed state) and to frame and forward this information onto the Client Manager for communication with the end-use CPE devices. This process is relatively straight-forward as communication flow forms a many-to-one scenario - multiple Child Processes send information to the single thread which queues individual notification for delivery to a single Client Manager.

The amount of traffic generated here is expected to be minimal, messages are only generated when a Child Process changes its classification of a flow, this is expected to occur infrequently for most flows. Also, no notification is ever sent if a flow is only ever classified as non-realtime - this should be the majority of flows.

An implementation would block on data to be received from any of the communications channels to the multiple Child Processes, as information is received from a Child, it is formatted for susbsequent delivery to the Client Manager.

3) Primary Thread: As well as launching the other threads to perform the management of the Flow Classifier, the Primary Thread is also responsible for periodically polling the database for updated configuration information. This includes configuration settings for both the Client Manager to forward Classification Changes to and the number of flows to allocate to individual Child Processes for handling. 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

B. Child Process

The Child Process is responsible for receiving per-flow packet statistics from the Parent Process, performing classification on those flows, and forwarding any changes in classification back to the Parent Process. The Child Process is designed to be independent of any other concurrent Child Processes and can be implemented to assume that only flows it receives data for are of interest. The Child Process should not assume the maximum number of flows to be handled, management of how many flows to allocate to a Child Process is handled by the Parent Process.

This independence allows for multi-hardware load balancing as the Child Process may be run on different hardware than the Parent Process. For each packet in each flow, the Child Process must calculate and update

per-flow statistics, which will then be used by a Classifier module to classify the flow. These tasks can be performed sequentially and do not require independent threads for implementation.

In our prototype, the classifier module is implemented as a Machine Learning Classifier, more details are provided later in this document.

Communications between the Parent and Child Processes are to be performed using a standard inter-process communication technique.

C. Inter-Process Communication

We use standard Interprocess Communication (IPC) techniques allow data to be passed between two separate processes on a system. Of the available IPC methods, we plan to use IPC messages and a Message Queue to share data between processes. These IPC Messages can be considered analogous to 'packets' (as used in IP socket communications).

We create a single, private message queue by the parent process when the system is launched. By making the queue private we ensure that only the parent and its child processes are allowed access.

As the queue is shared amongst the parent/child(ren), we need a means of addressing the intended recipient of each message. Also we would like to implement a scheme where children may only send messages to the parent - not to each other. This is implemented using the process PID of the intended destination process as the "message type" of the IPC Message field. This scheme works well as the parent knows the PID of all the Child Processes while the Child only knows the PID of itself and the parent. Messages are only removed from the queue by the process with PID matching the "message type" of the message.

The design of the ANGEL Flow Classifier makes extensive use of messages, one message is generated for EACH packet captured by a Flow Meter. A single message from the Flow Meter often contains statistics for approximately 100 captured packets. Also messages are generated for each change in classification - but this occurs less often. The system running the Flow Classifier **SHOULD** configure the IPC messages to support 1,024 or 2,048 system messages to ensure that threads and processes are not blocked due to a full message queue.

This common problem was discovered during implementation on a FreeBSD workstation. The default FreeBSD IPC configuration only allows for 16 pending messages to be placed in the message queue. Also the implementation of message queueing means that if a signal is received by the application from the OS while a process is blocked on the message queue, then the program is terminated.

In order to fix this problem on a FreeBSD system the following lines need to be added to the file `/boot/loader.conf`

```
kern.ipc.msgmnb=8192
kern.ipc.msgmni=40
kern.ipc.msgseg=1024
kern.ipc.msgssz=16
kern.ipc.msgtwl=2048
```

ANGEL is not the only application that makes extensive use of IPC messaging requiring this reconfiguration of the OS IPC Message System. The Squid Proxy Server [3] also requires reconfiguration of the underlying IPC message queues to improve performance.

D. Inter-Thread Communication

All threads need to be able communicate data amongst in 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 shared variables **must** be protected with a mutual exclusion.

E. Clustering Operating Systems

The Flow Classifier is designed in this way such that it might work with a clustered operating system such as OpenMosix. In this system management of

processes and inter-process communications is managed by the Operating System which provides unique process IDs across the cluster. Similarly pipes and other inter-process communication channels are managed by the Clustered kernel code to forward data between computers if required - this is all performed transparently to the applications.

The end result is that any program that is written to launch and create child processes may find those processes actually launched and executed on other computers within the cluster. A primary advantage of using the clustered operating system to perform load balancing is that the final code can execute unchanged on a single - non clustered - system.

F. Development Tools

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

IV. Parent Process

The parent process is the primary system process that executes when the Flow Classifier is instatiated. Its primary responsibities are to manage the Classifier framework such that classification work can be delegated to one of multiple Child Processes, each potentially running on different machines within a cluster.

In order to simplify implementation, the role of the Primary Process can be considered to be twofold:

1) Receive Packet statistics from the assorted Flow Meters and forward them to the appropriate Child Classification Processes to be classified
2) Receive classification information from the Child Processes and forward this to the Client Manager for ultimate delivery to relevant CPE devices

The implementation of this is broken into two threads of execution, with the primary thread maintaining overall system management and control.

A. Meter Network Thread

This thread should be implemented in a class called FlowMeterNetworkTask. The FlowMeterNetworkTask 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
- Manage creation and termination of independent Child Processes as required to classify packets for independent flows
- Manage mapping of flow hashes to Child Processes for classification purposes

Once running the main service thread should continuously:

- Get the next packet sent to it from one of the system Flow Meters
- Parse that packet and extract statistics for each individual packet
- Determine which Child Process is responsible handling the flow of each individual packet.
  - If no process is currently handling that flow (new flow) then assign the flow to a child process
  - If all child processes have been allocated a full quota of flows to handle, create a new child process to handle flows
- If the packet statistics signal that the flow has terminated
  - Remove the Flow Hash from the list of flows handled by a particular Child Process
  - Forward the flow termination signal to the Child Process
  - If the Child Process is no longer processing any active flows, signal the Child Process to terminate
- Pass the packet statistics to the Child Process for classification
- If signalled to terminate by the primary thread we need to stop reading received data, signal (and wait for) the Child Processes to terminate, and terminate the thread

1) Construction/Initialisation: The class constructor should create the UDP socket for receiving data from the Flow Meters. The socket should be closed in the destructor.

2) Main Processing: Processing is performed in the **svc()** method. This should run as an infinite loop which alternates between checking for packets arrived at the UDP Socket and for messages queued to its Message Queue. The only message sent to this Thread is the **MB_HANGUP** message to signal thread termination.

For any packets received on the UDP socket, they should be retrieved and parsed to extract flow hash information. Helper classes should be used to determine which Child Process to deliver the packet data to.

B. Manager Network Thread

This thread should be implemented in a class called ClientManagerNetworkTask. The ClientManagerNetworkTask 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

Once running the main service thread should continuously:

- Get the next flow classification signal sent by any of the Classifier Child Processes
- Format that signal into the appropriate form for delivery to the Client Manager
- Send the change in Flow State information to the Client Manager for ultimate delivery to appropriate CPE device(s)

1) Construction/Initialisation: The class constructor should create the UDP socket for later communications by the thread to the Client Manager. The socket should be closed in the destructor.

2) Main Processing: Processing is performed in the **svc()** method. This should run as an infinite loop which alternates between checking for signal received from the Child Processes and for messages queued to its Message Queue. The only message sent to this Thread is the **MB_HANGUP** message to signal thread termination.

For any signals received from the Child Processes, they should be reformatted for delivery to the Client Manager and sent over the UDP Socket.

C. Primary Thread

The primary responsibilities of the primary thread are:

- Initialise the Flow Classifier System
- Launch the Flow Meter Network Thread
- Launch the Client Manager Network Thread
- Periodically poll the Database for changes to the underlying system configuration
- Detect a system termination signal and signal the created threads to terminate
- Properly release resources upon termination

V. Parent Helper Classes

This section highlights some of the Helper Classes used by the Parent Process within the ANGEL Flow Classifier implementation. These classes are primarily used to manage internal data storage for the Classifier. The general functionality of the major classes have been described in previous sections.

The Parent Process must have some means for managing which Child Processes are allocated responsibility for classifying individual flows and for forwarding packet information to the correct Child Process. This is managed by the **ChildManager** and **FlowManager** classes.

A. ChildManager

The **ChildManager** class manages creation and destruction of child processes to perform the actual classification. It also manages a list of idle (or not fully allocated) processes and assigns a PID as requested when a new flow is created. This Class is used exclusively by the **FlowManager** class to allocate PIDs to indivudual flow hashes

The class should provide two methods, one to assign a PID of a child able to process another flow and one to decrement the count of flows managed by a child PID.

The first method would be called when a new flow is detected and a child process must be allocated to handle it. This method should return the PID of an existing child process with spare capacity or - if none exists - fork a new child process and return its PID.

The second method would be called when a flow has terminated and is no longer managed by a certain PID. This method should update internal counters for that child process to maintain the count of flows handled by that child. If the child is currently servicing no flows then the child process must be signalled to terminate.

This helper class is **only** used by the Meter Network Thread within the Flow Classifier, as such usage of Mutual Exclusions to protect changes to internal variables is not required.

B. FlowManager

The **FlowManager** class manages allocation, storage and mapping of Child Process PIDs to Flow Hashes. An instance of the **ChildManager** class is used to create, destroy and allocate Child Processes for this class. The class maintains an internal mapping of flow hashs to child process PIDs.

The class should provide two methods, one to get the PID of the child process that is handling the supplied Flow Hash and one to remove a Flow Hash from the map when the flow has terminated.

The first method would be called by the Meter Network Thread to determine the correct PID to forward the packet statistics to based on the Flow Hash. This method should return the current mapping of the specified hash to child process. If the flow is new and no mapping exists, then it should use the **ChildManager** instance

to allocate a PID (and fork a process if necessary) and update the map.

The second method would be called by the Meter Network Thread when a flow has terminated. This method should remove the current flow hash to PID mapping and use the **ChildManager** instance to decrement the count of flows handled by the process (and terminate the process if necessary).

This helper class is **only** used by the Meter Network Thread within the Flow Classifier, as such usage of Mutual Exclusions to protect changes to internal variables is not required.

VI. Child Process

This is implemented as an extra set of processes rather than threads to facilitate the implementation of load balancing scalability and reliabity through the use of a clustered-type operating system such as OpenMosix.

Each process is implemented such that it can receive IPC messages containing packet statistics from the Parent Process, the Parent sorting packets into unique flows and assigning a subset of flows to each Child Process. The overall implementation is such that a Child Process can operate independently of all other Child Processes - all packets for a particular flow will always be forwarded to the same Child Process.

Any changes in flow classification needs to be forwarded back to the Parent Process via an IPC message for further communication to the Client Manager. Child Processes also need to terminate cleanly when told to by the Parent Process (no more flows to manage).

The ANGEL Prototype implementation should use a Machine Learning based system to perform flow classification.

The process should be implemented in a class called ClassifierProcess. The ClassifierProcess class should:

- Receive packet statistic information from the shared IPC message queue with the Parent Process
- Keep track of all flows - using the Flow Hash - from packet statistics it has seen
- Store and update statistics for each flow as a packet arrives for that flow
- Classification is done on a sliding/jumping window basis. When the window for a particular flow is full, that flow should be classified and all statistics for that flow should be zeroed for subsequent classification on the next window of packets
- Classification is performed using some form of Machine Learning technique
- When the classification of a flow has changed, signal the parent process

Below we discuss the key aspects as relating to the design of key components for this process

A. Per Flow Statistics

The process needs to uniquely track all packets for all flows assigned it by the Parent Process. This information **should** be stored in some form of internal database.

If possible - based on the Machine Learning implementation - we should calculate as many flow statistics as possible on a packet-by-packet basis. This minimises requirements for storing and indexing large amounts amounts of data as well as spread the processing load away from points in time when the sliding window is full.

B. Sliding/Jumping Window

It is essential that Classification is performed while the Flow is still active, this allows ANGEL to enable flow prioritisation while it may be beneficial. Use of a sliding window allows ANGEL to classify a flow once enough packets for a flow have been seen. The concept of flow classification using a sliding window was first demonstrated by Nguyen [4], [5].

A true sliding window implementation would perform a classification on the last N packets seen for each flow. Once a flow initially fills the sliding window, it would be possible to re-determine the flow classification for each new packet seen on that flow. This would also result in a high processing load due to repeated classifications for each single packet witnessed. There is also the increased complexity in managing the statistic calculation that would need to be performed in this case - removing the statistics for the packet that is no longer within the window and adding the statistics of the newest packet.

The compromise position adopted in the ANGEL Flow Classifier is to use a jumping window. In this case once the jumping window is full with N packets, a classification is made. The window is then emptied and a classification is no longer made until another N packets are witnessed. This results in one classification attempt for every N packets seen on a particular flow. The effect of this is a jumping window where the window jumps its full size to process the next set of packets.

C. Classification

The prototype ANGEL Flow Classifier performs classification using a machine learning (ML) algorithm and a vector of flow statistics calculated from the previous N

packets received for each flow where *N* is the size of the jumping/sliding window. The result of the classification indicates whether the flow is realtime or non-realtime with the ultimate aim being whether the ANGEL CPE devices should attempt to prioritise (realtime) a flow or provide the standard best-effort services (non-realtime).

There are numerous ML algorithms that can be used to provide classification. In our prototype we use an implementation is of a Naive Bayes Classifier. The ML algorithm is able to predict the class of a flow using a classification model that is loaded from a configuration file when the application is launched.

D. Classification Defaults

To simplify implementation of the code that signals the Parent Process, the default classification of a flow is for **Non-Realtime** traffic. In this case:

- If the first classification is **non-realtime** then the classification is unchanged and the Parent Process is not signalled - the desired result
- If the first classification is **realtime** then the classification is changed and the Parent Process is signalled with the fact that the flow is realtime

E. Minimising Flapping

In order to minimise the potential for classification flapping (one window is classified as realtime, the next as non-realtime, then realtime, etc.), we use an approach we call *"Confirmed Classification"*. This technique requires that a classification be confirmed before updating the actual classification and signalling the parent process. In effect, there must be two consecutive **realtime** classifications for the classification to change from **non-realtime** to **realtime**. Similarly, the must be two consecutive **non-realtime** classifications for the classification to change from **realtime** to **non-realtime**.

The basic algorithm deployed in the Classifier is:

```
loop
{
  get packet stats

  if new flow
    create flow stats
    flow[classification] = flow[last] = non-realtime

  add packet to flow
  if flow window full
    current = classify flow stats

    if (current = flow[last]) AND (current != flow[classification])
      flow[classification] = current
      signal parent of new classification - current

    flow[last] = current
    clear flow stats
}
```

VII. Child Helper Classes

This section highlights some of the Helper Classes used by the Child Process within the ANGEL Flow Classifier implementation. These classes are primarily used to manage internal data storage for the Classifier. The general functionality of the major classes have been described in previous sections.

The Child Process must have some means for managing calculation of flow statistics for each flow for which it is responsible, as well as maintaining and managing this information for all flows. This is managed by the **Flow** and **FlowMap** classes. A further responsibility is to perform the actual classification of flows, this is achieved with the **NaiveBayesClassifier** class.

A. Flow

This Class is used within the Child Process to track the statistics being calculated for each flow. Individual packet statistics are *added* to this Class instance in a manner that updates internal values. The Class is also able to return flow statistics the calling Child Process and to clear the statistics to signify the start of a new classification window.

B. FlowMap

As each Child Process is responsible for classifying a number of flows, there is a requirement for a data structure to keep track of all the instances of the **Flow** class. This data structure stores the mapping of Flow Hash to Flow instances within the Child Process.

The class is implemented as a Singleton class within the Child Process which manages thread-safe access to an STL map data structure to store the required information.

C. NaiveBayesClassifier

The ML Naive Bayes Classifier is implemented within the **NaiveBayesClassifier** class. This implementation is configured with a classification model which is loaded from a configuration file when the Flow Classifier application is launched. A change in the classification model will require the Classifier (for our prototype implementation) to be stopped and re-started.

The Classifier returns the classification of a flow based on statistics gathered from the packet statistics of the previous *N* packets for a given flow. As per the previous section, the result it used by the Child Process to determine whether the classification for a flow has changed and whether to signal this change to the Parent Process.

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] OpenMosix Community, "The OpenMosix Project," January 2007, http://openmosix.sourceforge.net.

[3] Squid Open Source Development Community, "Squid Web Proxy Cache," January 2007, http://www.squid-cache.org/.

[4] T. Nguyen and G. Armitage, "Training on multiple sub-flows to optimise the use of Machine Learning classifiers in real-world IP networks," in *Proceedings of the IEEE 31st Conference on Local Computer Networks* , Florida, USA, 2006.

[5] T. Nguyen and G. Armitage, "Synthetic Sub-flow Pairs for Timely and Stable IP Traffic Identification," in *Proceedings of the Australian Telecommunication Networks and Application Conference* , Melbourne, Australia, 2006.