Traffic Flow Measurement: Experiences with NeTraMet

Status of this Memo

Abstract

   This memo records experiences in implementing and using the Traffic
   Flow Measurement Architecture and Meter MIB. It discusses the
   implementation of NeTraMet (a traffic meter) and NeMaC (a combined
   manager and meter reader), considers the writing of meter rule sets
   and gives some guidance on setting up a traffic flow measurement
   system using NeTraMet.

Table of Contents

1 Introduction

   Early in 1992 my University needed to develop a system for recovering
   the costs of its Internet traffic. In March of that year I attended
   the Internet Accounting Working Group's session at the San Diego
   IETF, where I was delighted to find that the Group had produced a
   detailed architecture for measuring network traffic and were waiting
   for someone to try implementing it.

   During 1992 I produced a prototype measurement system, using balanced
   binary trees to store information about traffic flows. This work was
   reported at the Washington IETF in November 1992. The prototype
   performed well, but it made no attempt to recover memory from old
   flows, and the overheads in managing the balanced trees proved to be
   unacceptably high. I moved on to develop a production-quality
   system, this time using hash tables to index the flow information.

   This version was called NeTraMet (the Network Traffic Meter), and was
   released as free software in October 1993. Since then I have
   continued working on NeTraMet, producing new releases two or three
   times each year. NeTraMet is now in production use at many sites
   around the world. It is difficult to estimate the number of sites,
   but there is an active NeTraMet mailing list, which had about 130
   subscribers in March 1996.

   Early in 1996 the Realtime Traffic Flow Measurement Working Group
   (RTFM) was chartered to move the Traffic Flow Measurement
   Architecture on to the IETF standards track. This document records
   traffic flow measurement experience gained through three years
   experience with NeTraMet.

1.1 NeTraMet structure and development

   The Traffic Flow Architecture document [1] describes four components:

     - METERS, which are attached to the network at the points where
       it is desired to measure the traffic,

     - METER READERS, which read data from meters and store it for later
       use,

     - MANAGERS, which configure meters and control meter readers, and

     - ANALYSIS APPLICATIONS, which process the data from meter readers
       so as to produce whatever reports are required.

   NeTraMet is a computer program which implements the Traffic Meter,
   stores the measured flow data in memory, and provides an SNMP agent
   so as to make it available to Meter Readers. The NeTraMet
   distribution files include NeMaC, which is a combined Manager and
   Meter Reader capable of managing an arbitrary number of meters, each
   of which may be using its own rule set, and having its flow data
   collected at its own specified intervals. The NeTraMet distribution
   also includes several rudimentary Analysis Applications, allowing
   users to produce simple plots from NeMaC's flow data files (fd_filter
   and fd_extract) and to monitor - in real time - the flows at a remote
   meter (nm_rc and nifty).

   Since the first release the Traffic Meter MIB [2] has been both
   improved and simplified. Significant changes have included better
   ways to specify traffic flows (i.e. more actions and better control
   structures for the Packet Matching Engine), and computed attributes
   (class and kind). These changes have been prompted by operational
   requirements at sites using NeTraMet, and have been tested
   extensively in successive versions of NeTraMet.

   NeTraMet is widely used to collect usage data for Internet Service
   Providers. This is especially so in Australia and New Zealand, but
   there are also active users at sites around the world, for example in
   Canada, France, Germany and Poland.

   NeTraMet is very useful as a tool for understanding exactly where
   traffic is flowing in large networks. Since the Traffic Meters
   perform considerable data reduction (as specified by their rule sets)
   they significantly reduce the volume of data to be read by Meter
   Readers. This characteristic makes NeTraMet particularly effective
   for networks with many remote sites. An example of this (the
   Kawaihiko network) is briefly described below.

As well as providing data for post-observation analysis, NeTraMet can
be used for real-time network monitoring and trouble-shooting. The
NeTraMet distribution includes 'nifty,' an X/Motif application which
monitors traffic flows and attempts to highlight those which are
'interesting.'

1.2 Scope of this document

This document presents the experience gained from three years work
with the Traffic Flow Measurement Architecture. Its contents are
grouped as follows

   - Implementation issues for NeTraMet and NeMaC,

   - How rule files work, and how to write them for particular
     purposes, and

   - How to use NeTraMet and NeMaC for short-term and long-term flow
     measurement.

2 Implementation

2.1 Choice of meter platform

As pointed out in the Architecture document [1], the goal of the
Realtime Traffic Flow Measurement Working Group is to develop a
standard for the Traffic Meter, with the goal of seeing it
implemented in network devices such as hubs, switches and routers.
Until the Architecture is well enough developed to allow this, it has
sufficed to implement the meter as a program running on a general-
purpose computer system.

The choice of computer system for NeTraMet was driven by the need to
choose one which would be widely available within the Internet
community. One strong possibility was a Unix system, since these are
commonly used for a variety of network support and management tasks.
For the initial implementation, however, Unix would have had some
disadvantages:

  - The wide variety of different Unix systems can increase the
    difficulties of software support.

  - The cost of a Unix system as a meter is too high to allow users
    to run meters simultaneously at many points within their
    networks.

Another factor in choosing the platform was system performance. When
I first started implementing NeTraMet it was impossible to predict
how much processing workload was needed for a viable meter.
Similarly, I had no idea how much memory would be required for code
or data. I therefore chose to implement NeTraMet on a DOS PC. This
was because:

  - It is a minimum system in all respects. If the meter works
    well on such a system, it can be implemented on almost any
    hardware (including routers, switches, etc.)

  - It is an inexpensive system. Sites can easily afford to have
    many meters around their networks.

  - It is a simple system, and one which I had complete control over.
    This allowed me to implement effective instrumentation to monitor
    the meter's performance, and to include a wide variety of
    performance optimisations in the code.

Once the meter was running I needed a manager to download rule files
to it. Since a single manager and meter reader can effectively
support a large number of meters, a Unix environment for NeMaC was a
natural choice. There are fewer software support problems for NeMaC
than for NeTraMet since NeMaC has minimal support needs - it only
needs to open a UDP socket to the SNMP port on each controlled meter.

Early NeTraMet distributions contained only the PC meter and Unix
manager. In later releases I ported NeTraMet (the meter) to Unix,
and extended the control features of NeMaC (the combined manager and
meter reader). I have also experimented with porting NeMaC to the
DOS system. This is not difficult, but doesn't seem to be worth
pursuing.

The current version of NeTraMet is a production-quality traffic
measurement system which has been in continuous use at the University
of Auckland for nearly two years.

2.2 Programming support requirements

To implement the Traffic Flow Meter I needed a programming
environment providing good support for the following:

  - observation of packet headers on the network;

  - system timer with better than 10 ms resolution;

  - IP (Internet Protocol), for communications with manager and meter
    reader;

   - SNMP, for the agent implementing the Meter MIB.

2.2.1 DOS environment

   For the PC I chose to use Ethernet as the physical network medium.
   This is simply an initial choice, being the medium used within the
   University of Auckland's data network. Interfaces for other media
   could easily be added as they are needed.

   In the PC environment a variety of 'generalised' network interfaces
   are available. I considered those available from companies such as
   Novell, DEC and Microsoft and decided against them, partly because
   they are proprietary, and partly because they did not appear to be
   particularly easy to use. Instead I chose the CRYNWR Packet Drivers
   [3] . These are available for a wide variety of interface cards and
   are simple and clearly documented. They support Ethernet's
   promiscuous mode, allowing one to observe headers for every passing
   packet in a straightforward manner. One disadvantage of the Packet
   Drivers is that it is harder to use them with newer user shells (such
   as Microsoft Windows), but this was irrelevant since I intended to
   run the meter as the only program on a dedicated machine.

   Timing on the PC presented a challenge since the BIOS timer routines
   only provide a clock tick about 18 times each second, which limits
   the available time resolution. Initially I made do with a timing
   resolution of one second for packets, since I believed that most
   flows existed for many seconds. In recent years it has become
   apparent that many flows have lifetimes well under a second. To
   measure them properly with the Traffic Flow Meter one needs times
   resolved to 10 millisecond intervals, this being the size of
   TimeTicks, the most common time unit within SNMP [4]. Since all the
   details of the original PC are readily available [5], it was not
   difficult to understand the underlying hardware. I have written PC
   timer routines for NeTraMet which read the hardware timer with 256
   times the resolution of the DOS clock ticks, i.e. about 5 ticks per
   millisecond.

   There are many TCP/IP implementations available for DOS, but most of
   them are commercial software. Instead I chose Waterloo TCP [6],
   since this was available (including full source code) as public
   domain software. This was necessary since I needed to modify it to
   allow me to save incoming packet headers at the same time as
   forwarding packets destined for the meter to the IP handler routines.
   For SNMP I chose CMU SNMP [7], since again this was available (with
   full source code) as public domain software. This made it fairly
   simple to port it from Unix to the PC.

Finally, for the NeTraMet development I used Borland's Turbo C and
Turbo Assembler. Although many newer C programming environments are
now available, I have been perfectly happy with Turbo C version 2 for
the NeTraMet project!

2.2.2 Unix environment

In implementing the Unix meter, the one obvious problem was 'how do I
get access to packet headers?' Early versions of the Unix meter were
implemented using various system-specific interfaces on a SunOS 4.2
system. Later versions use libpcap [8], which provides a portable
method of obtaining access to packet headers on a wide range of Unix
systems. I have verified that this works very well for ethernet
interfaces on Solaris, SunOS, Irix, DEC Unix and Linux, and for FDDI
interfaces on Solaris. libpcap provides timestamps for each packet
header with resolution determined by the system clock, which is
certainly better than 10 ms!

All Unix systems provide TCP/IP capabilities, so that was not an
issue. For SNMP I used CMU SNMP, exactly as on the PC.

2.3 Implementing the meter

This section briefly discusses the data structures used by the meter,
and the packet matching process. One very strong concern during the
evolution of NeTraMet has been the need for the highest possible
level of meter performance. A variety of interesting optimisations
have been developed to achieve this; as discussed below. Another
particular concern was the need for efficient and effective memory
managent; this is discussed in detail below.

2.3.1 Data structures

All the programs in NeTraMet, NeMaC and their supporting utility
programs are written in C, partly because C and its run-time
libraries provides good access to the underlying hardware, and partly
because I have found it to be a highly portable language.

The data for each flow is stored in a C structure. The structure
includes all the flow's attribute values (including packet and byte
counts), together with a link field which can be used to link flows
into lists. NeTraMet assumes that Adjacent addresses are 802 MAC
Addresses, which are all six bytes long. Similarly, Transport
addresses are assumes to be two bytes long, which is the case for
port numbers in IP. Peer addresses are normally four bytes or less
in length. They may, however, be as long as 20 bytes (for CLNS). I
have chosen to use a fixed Peer address size, defined at compile
time, so as to avoid the complexity of having variable-sized flow
structures.

The flow table itself is an array of pointers to flow data
structures, which allows indexed access to flows via their flow
numbers. There is also a single large hash table, referred to in the
Architecture document [1] as the flow table's 'search index'. Each
hash value in the table points to a circular chain of flows. To find
a flow one computes its hash value then searches that value's flow
chain.

The meter stores each rule in a C structure. All the rule components
have fixed sizes, but address fields must be wide enough to hold any
type of address - Adjacent, Peer or Transport. The rule address
width is defined at compile time, in the same way as flow Peer
addresses. Each rule set is implemented as an array of pointers to
rule data structures, and the rule table is an array of pointers to
the rule sets. The size of each rule set is specified by NeMaC
(before it begins downloading the rule set), but the maximum number
of rule sets is defined at compile time.

2.3.2 Packet matching

Packet matching is carried out in NeTraMet exactly as described in
the Architecture document [1]. Each incoming packet header is
analysed so as to determine its attribute values. These values are
stored in a structure which is passed to the Packet Matching Engine.
To facilitate matching with source and destination reversed this
structure contains two substructures, one containing the source
Adjacent, Peer and Transport address values, the other containing the
destination address values.

2.3.3 Testing groups of rule addresses

As described in the Architecture [1] each rule's address will usually
be tested, i.e. ANDed with the rule's mask and compared with the
rule's value. If the comparison fails, the next rule in sequence is
executed. This allows one to write rule sets which use a group of
rules to test an incoming packet to see whether one of its addresses

   - e.g. its SourcePeerAddress - is one of a set of specified IP
   addresses. Such groups of related rules can grow quite large,
   containing hundreds of rules. It was clear that sequential execution
   of such groups of rules would be slow, and that something better was
   essential.

   The optimisation implemented in NeTraMet is to find groups of rules
   which test the same attribute with the same mask, and convert them
   into a single hashed search of their values. The overhead of setting
   up hash tables (one for each group) is incurred once, just before the
   meter starts running a new rule set. When a 'group' test is to be
   performed, the meter ANDs the incoming attribute value, computes a
   hash value from it, and uses this to search the group's hash table.
   Early tests showed that the rule hash chains were usually very short,
   usually having only one or two members. The effect is to reduce
   large sequences of tests to a hash computation and lookup, with a
   very small number of compares; in short this is an essential
   optimisation for any traffic meter!

   There is, of course, overhead associated with performing the hashed
   compare. NeTraMet handles this by having a minimum group size
   defined at compile time. If the group is too small it is not
   combined into a hashed group.

   In early versions of NeTraMet I did not allow Gotos into a hashed
   group of rules, which proved to be an unnecessarily conservative
   position. NeTraMet stores each group's hash table in a separate
   memory area, and keeps a pointer to the hash table in the first rule
   of the group. (The rules data structure has an extra component to
   hold this hash table pointer). Rules within the group don't have
   hash table pointers; when they are executed as the target of a Goto
   rule they behave as ordinary rules, i.e. their tests are performed
   normally.

2.3.4 Compression of address masks

   When the Packet Matching Engine has decided that an incoming packet
   belongs to a flow which is to be measured, it searches the flow table
   to determine whether or not the flow is already present. It does
   this by computing a hash from the packet and using it for access to
   the flow table's search index.

   When designing a hash table, one normally assumes that the objects in
   the table have a constant size. For NeTraMet's flow table this would
   mean that each flow would contain a value for every attribute. This,
   however, is not the case, since only those attribute values 'pushed'
   by rules during packet matching are stored for a flow.

To demonstrate this problem , let us assume that every flow in the
table contains a value for only one attribute, SourcePeerAddress, and
that the rule set decides whether flows belong to a specified list of
IP networks, in which case only their network numbers are pushed.
The rules perform this test using a variety of masks, since the
network number allocations range from 16 to 24 bits in width. In
searching the flow table, the meter must distinguish between zeroes
in the address and 'don't care' bits which had been ANDed out. To
achieve this it must store SourcePeerMask values in the flow table as
well as the ANDed SourcePeerAddress values.

In early versions of NeTraMet this problem was side-stepped by using
multiple hash tables and relying on the user to write rules which
used the same set of attributes and masks for all the flows in each
table. This was effective, but clumsy and difficult to explain.
Later versions changed to using a single hash table, and storing the
mask values for all the address attributes in each flow.

The current version of the meter stores the address masks in
compressed form. After examining a large number of rule sets I
realised that although a rule set may have many rules, it usually has
a very small number of address masks. It is a simple matter to build
a table of address masks, and store an index to this 'mask table'
instead of a complete mask. NeTraMet's maximum number of masks is
defined at compile time, up to a maximum of 256. This allows me to
use a single byte for each mask in the flow data structure,
significantly reducing the structure's size. As well as this size
reduction, two masks can be compared by comparing their indices in
the mask table, i.e. it reduces to a single-byte comparison.
Overall, using a mask table seems to provide useful improvements in
storage efficiency and execution speed.

2.3.5 Ignoring unwanted flow data

As described in the Architecture document [1], every incoming packet
is tested against the current rule set by the Packet Matching Engine.
This section explains my efforts to improve NeTraMet performance on
the PC by reducing the amount of processing required by each incoming
packet.

On the PC each incoming packet causes an interrupt, which NeTraMet
must process so as to collect information about the packet. In early
versions I used a ring buffer with 512 slots for packet headers, and
simply copied each packet's first 64 bytes into the next free slot.
The packet headers were later taken from the buffer, attribute values
were extracted from them, and the resulting 'incoming attribute
values' records were passed to the Packet Matching Engine.

I modified the interrupt handling code to extract the attribute
values and store them in a 'buffer slot.' This reduced the amount of
storage required in each slot, allowing more space for storing flows.
It did increase slightly the amount of processing done for each
packet interrupt, but this has not caused any problems.

In later versions I realised that if one is only interested in
measuring IP packets, there is no point in storing (and later
processing) Novell or EtherTalk packets! It is a simple matter for
the meter to inspect a rule set and determine which Peer types are of
interest. If there are PushRule rules which test SourcePeerType (or
DestPeerType), they specify which types are of interest. If there
are no such rules, every packet type is of interest. The PC NeTraMet
has a set of Boolean variables, one for each protocol it can handle.
The values of these 'protocol' variables are determined when the
meter begins running a new rule set. For each incoming packet, the
interrupt handler determines the Peer type. If the protocol is not
of interest, no further processing is done - the packet is simply
ignored. In a similar manner, if Adjacent addresses are never tested
there is no point in copying them into the packet buffer slot.

The overall effect of these optimisations is most noticeable for rule
files which measure IP flows on a network segment which also carries
a lot of traffic for other network protocols; this situation is
common on multiprotocol Local Area networks. On the Unix version of
NeTraMet the Operating System does all the packet interrupt
processing, and libpcap [8] delivers packet headers directly to
NeTraMet. The 'protocol' and 'adjacent address' optimisations are
still performed, at the point when NeTraMet receives the packet
headers from libpcap.

2.3.6 Observing meter reader activity

The Architecture document [1] explains that a flow data record must
be held in the meter until its data has been read by a meter reader.
A meter must therefore have a reliable way of deciding when flow data
has been read. The problem is complicated by the fact that there may
be more than one meter reader, and that meter readers collect their
data asynchronously.

Early versions of NeTraMet solved this problem by having a single MIB
variable which a meter reader could set to indicate that it was
beginning a data collection. In response to such an SNMP SET
request, NeTraMet would update its 'collectors' table. This had an
entry for each meter reader, and variables recording the start time
for the last two collections. The most recent collection might still
be in progress, but its start time provides a safe estimate of the
time when the one before it actually finished. Space used for flows
which have been idle since the penultimate collection started can be
recovered by the meter's garbage collector, as described below.

The Meter MIB [2] specifies a more general table of meter reader
information. A meter reader wishing to collect data from a meter
must inform the meter of its intention by creating a row in the
table, then setting a LastTime variable in that row to indicate the
start of a collection. The meter handles such a SET request exactly
as described above. If there are multiple meter readers the meter
can easily find the earliest time any of them started its penultimate
collection, and may recover flows idle since then. Should a meter
reader fail, NeTraMet will eventually time out its entry in the meter
reader info table, and delete it. This avoids a situation where the
meter can't recover flows until they have been collected by several
meter readers, one of which has failed.

2.3.7 Meter memory management

In principle, the size of the flow table (i.e. the maximum number of
flows) could be changed dynamically. This would involve allocating
space for the flow table's new pointer array and copying the old
pointers into it. NeTraMet does not implement this. Instead the
maximum number of flows is set from the command line when it starts
execution. If no maximum is specified, a compile-time default number
is used.

Memory for flow data structures (i.e. 'flows') is allocated
dynamically. NeTraMet requests the C run-time system for blocks of
several hundred flows, and links them into a free list. When a new
flow is needed NeTraMet gets memory space from the free list, then
searches the flow table's pointer array for an unused flow pointer.
In practice a 'last-allocated' index is used to point to the flow
table, so a simple linear search suffices. The flow index is saved
in the flow's data record, and its other attribute values are set to
zero.

To release a flow data record it must first be removed from any hash
list it is part of - this is straightforward since those lists are
circular. The flow's entry in the flow table pointer array is then
set to zero (NULL pointer), and its space is returned to the free
list.

Once a flow data record is created it could continue to exist
indefinitely. In time, however, the meter would run out of space.
To deal with this problem NeTraMet uses an incremental garbage
collector to reclaim memory.

At regular intervals specified by a 'GarbageCollectInterval' variable
the garbage collector procedure is invoked. This searches through
the flow table looking for flows which might be recovered. To
control the resources consumed by garbage collection there are limits
on the number of in-use and idle flows which the garbage collector
may inspect these are set either when NeTraMet is started (as
options on the command line) or dynamically by NeMaC (using variables
in an Enterprise MIB for NeTraMet)

To decide whether a flow can be recovered, the garbage collector
considers how long it has been idle (no packets in either direction),
and when its data was last collected. If it has been collected by
all known meter readers since its LastTime, its memory may be
recovered. This alogrithm is implemented using a variable called
'GarbageCollectTime,' which normally contains the meter's UpTime when
the penultimate collection (i.e. the one before last) was started.
See the section on observing meter reader activity (above) for more
details.

Should flows not be collected often enough the meter could run out of
space. NeTraMet attempts to prevent this by having a low-priority
background process check the percentage of flows active and compare
it with the HighWaterMark MIB variable. If the percentage of active
flows is greater than the high-water mark, 'GarbageCollectTime' is
incremented by the current value of the InactivityTimeout MIB
variable.

The Meter MIB [2] specifies that a meter should switch to using a
'standby' rule set if the percentage of active flows rises above
HighWaterMark. In using NeTraMet to measure traffic flows to and
from the University of Auckland it has not been difficult to create
standby rules which are very similar to the 'production' rule file,
differing only in that they push much less information about flows.
This has, on several occasions, allowed the meter to continue running
for one or two days after the meter reader failed. When the meter
reader restarted, it was able to collect all the accumulated flow
data!

The MIB also specifies that the meter should take some action when
the active flow percentage rises above its FloodMark value. If this
were not done, the meter could spend a rapidly increasing proportion
of its time garbage collecting, to the point where its ability to
respond to requests from its manager would be compromised. NeTraMet
switches to the default rule set when its FloodMark is reached.

A potentially large number of new flows may be created when the meter
switches to a standby rule set. It is important to set a
HighWaterMark so as to allow enough flow table space for this. In
practice, a HighWaterMark of 65% and a FloodMark of 95% seem to work
well.

2.4 Data collection

As explained above, a meter reader wishing to collect flows begins
each collection by setting the LastTime variable in its
ReaderInfoTable row, then works its way through the flow table
collecting data. A number of algorithms can be used to examine the
flow table; these are presented below.

The simplest approach is a linear scan of the table, reading the
LastTime variable for each row. If the read fails the row is
inactive. If it succeeds, it is of interest if its LastTime value is
greater than the time of the last collection. Although this method
is simple it is also rather slow, requiring an SNMP GET request for
every possible flow; this renders it impractical.

Early versions of NeTraMet used two 'windows' into the flow table to
find flows which were of interest. Both windows were SNMP tables,
indexed by a variable which specified a time. A succession of
GETNEXT requests on one of these windows allowed NeMaC (the meter
reader) to find the flow indices for all flows which had been active
since the specified time. The two windows were the ActivityTime
window (which located active flows), and the CreateTime window (which
located new flows). Knowing the index of an active flow, the meter
reader can GET the values for all the attributes of interest. NeMaC
allows the user to specify which these are, rather than simply read
all the attributes.

Having the two windows allowed NeMaC to read attributes which remain
constant - such as the flow's address attributes - when the flow is
created, but to only read attributes which change with time - such as
its packet and byte counts - during later collections. Experience
has shown, however, that many flows have rather short lifetimes; one
effect of this is that the improved efficiency of using two windows
does not result in any worthwhile improvement in collection
performance.

The current version of the Meter MIB [2] uses a TimeFilter variable
in the flow table entries. This can be used with GETNEXT requests to
find all flows which have been active since a specified time
directly, without requiring the extra 'window' SNMP variables. It
can be combined with SNMPv2's GETBULK request to further reduce the
number of SNMP packets needed for each collection; I have yet to
implement this in NeTraMet.

A disadvantage of using SNMP to collect data from the meter is that
SNMP packets impose a high overhead. For example, if we wish to read
an Integer32 variable (four bytes of data), it will be returned with
its object identifier, type and length, i.e. at least ten bytes of
superfluous data. One way to reduce this overhead is to use an
Opaque object to return a collection of data. NeTraMet uses this
approach to retrieve 'column activity data' from the meter, as
follows.

Each packet of column activity data contains data values for a
specified attribute, and each value is preceded by its flow number.
The flow table can be regarded as a two-dimensional array, with a
column for each flow attribute. Column activity data objects allow
the meter reader to read columns of the flow table, so as to collect
only those attributes specified by the user. The actual
implementation is complicated by the fact that since the flow table
is read column by column, rows can become active after the first
column has been read. NeMaC reads the widest columns (those with
greatest size in bytes, e.g. PeerAddress) first, and ignores any rows
which appear in later columns. Newly active rows will, of course, be
read in the next collection.

Using Opaque objects in this way dramatically reduces the number of
SNMP packets required to read a meter. This has proved worthwhile in
situations where the number of flows is large (for example on busy
LANs), and where the meter(s) are physically dispersed over slow WAN
links. It has the disadvantage that general-purpose MIB browsers
cannot understand the column activity variables, but this seems a
small price to pay for the improved data collection performance.

2.5 Restarting a meter

If a meter fails, for example because of a power failure, it will
restart and begin running rule set 1, the default rule set which is
built into the meter. Its manager must recognise that this has
happened, and respond with some suitable action.

NeMaC allows the user to specify a 'keepalive' interval. After every
such interval NeMaC reads the meter's sysUptime and compares it with
the last sysUptime. If the new sysUptime is less than the last one,

NeMaC decides that the meter has restarted. It downloads the meter's
backup rule set and production rule set, then requests the meter to
start running the production rule set. In normal use we use a
keepalive interval of five minutes and a collection interval of 15
minutes. If a meter restarts, we lose up to five minutes data before
the rules sets are downloaded.

Having the meter run the default rule set on startup is part of the
Traffic Flow Measurement Architecture [1], in keeping with the notion
that meters are very simple devices which do not have disk storage.
Since disks are now very cheap, it may be worth considering whether
the architecture should allow a meter to save its configuration
(including rule sets) on disk.

2.6 Performance

The PC version of the meter, NeTraMet, continually measures how much
processor time is being used. Whenever there is no incoming packet
data to process, 'dummy' packets are generated and placed in the
input buffer. These packets are processed normally by the Packet
Matching Engine; they have a PeerType of 'dummy.' The numbers of
dummy and normal packets are counted by the meter; their ratio is
used as an estimate of the processor time which is 'idle,' i.e. not
being used to process incoming packets. The Unix version is intended
to run as a process in a multiprocessing system, so it cannot busy-
wait in this way.

The meter also collects several other performance measures; these can
be displayed on the meter console in response to keyboard requests.

The PC meter can be used with a 10 MHz 286 machine, on which it can
handle a steady load of about 750 packets per second. On a 25 MHz
386SX it will handle about 1250 packets per second. Users have
reported that a 40 MHz 486 can handle peaks of about 3,000 packets
per second without packet loss. The Unix meter has been tested
metering traffic on a (lightly loaded) FDDI interface; it uses about
one percent of the processor time on a SPARC 10 system running
Solaris.

3 Writing rule sets

The Traffic Meter provides a versatile device for measuring a user-
specified set of traffic flows, and performing useful data reduction
on them. This data reduction capability not only minimises the
volume of data to be collected by meter readers, but also simplifies
the later processing of traffic flow data.

   The flows of interest, and the processing to be performed, are
   specified in a 'rule set' which is downloaded to the meter (NeTraMet)
   by the manager (NeMaC). This section explains what is involved in
   writing rule sets.

   NeTraMet is limited to metering packets observed on a network
   segment. This means that for all the observed flows, Source and Dest
   Type attributes (e.g. SourcePeerType and DestPeerType) have the same
   value.

   The NeTraMet implementation uses single variables in its flow data
   structure for AdjacentType, SourceType and TransType. Nonetheless,
   the rule sets discussed below push values for both Source and Dest
   Type attributes; this make sure that packet matching works properly
   with the directions reversed, even for a meter which allows Source
   and Dest Type values to be different.

3.1 Rule set to observe all flows

   NeMaC reads rule sets from text files which contain the rules, the
   set number which the meter (and meter reader) will identify them by,
   and a 'format,' i.e. a list specifying which attributes the meter
   reader should collect and write to the flow data file. The #
   character indicates the start of a comment; NeMaC ignores the rest of
   the line.

```
     SET 2
     #
     RULES
     #
       SourcePeerType & 255 = Dummy:    Ignore, 0;
       Null & 0 = 0:  GotoAct, Next;
     #
       SourcePeerType     & 255               = 0: PushPkttoAct, Next;
       DestPeerType       & 255               = 0: PushPkttoAct, Next;
       SourcePeerAddress  & 255.255.255.255 = 0: PushPkttoAct, Next;
       DestPeerAddress    & 255.255.255.255 = 0: PushPkttoAct, Next;
       SourceTransType    & 255               = 0: PushPkttoAct, Next;
       DestTransType      & 255               = 0: PushPkttoAct, Next;
       SourceTransAddress & 255.255           = 0: PushPkttoAct, Next;
       DestTransAddress   & 255.255           = 0: CountPkt, 0;
     #
     FORMAT FlowRuleSet FlowIndex FirstTime "  "
        SourcePeerType SourcePeerAddress DestPeerAddress "  "
        SourceTransType SourceTransAddress DestTransAddress "  "
        ToPDUs FromPDUs "  " ToOctets FromOctets;
```

The first rule tests the incoming packet's SourcePeerType to see
whether it is 'dummy.' If it is, the packet is ignored, otherwise
the next rule is executed.

The second rule tests the Null attribute. Such a test always
succeeds, so the rule simply jumps to the action of the next rule.
(The keyword 'next' is converted by NeMaC into the number of the
following rule.)

The third rule pushes the packet's SourcePeerType value, then jumps
to the action of the next rule. The user does not know in advance
what the value of PushPkt rules will be, which is why the value
appearing in them is always zero. The user must take care not to
write rule sets which try to perform the test in a PushPkt rule.
This is a very common error in a rule set, so NeMaC tests for it and
displays an error message.

The following rules push a series of attribute values from the
packet, and the last rule also Counts the packet, i.e. it tells the
Packet Matching Engine (PME) that the packet has been successfully
matched. The PME responds by searching the flow table to see whether
the flow is already current (i.e. in the table), creating a new flow
data record for it should this be necessary, and incrementing its
packet and byte counters.

Overall this rule set simply classifies the packet (i.e. decides
whether or not it is to be counted), then pushes all the Peer and
Transport attribute values for it. It makes no attempt to specify a
direction for the flow - this is left to the PME, as described in
[1]. The resulting flow data file will have each flow's source and
destination addresses in the order of the first packet the meter
observed for the flow.

3.2 Specifying flow direction, using computed attributes

As indicated above, the Packet Matching Engine will reliably
determine the flow, and the direction within that flow, for every
packet seen by a meter. If the rule set does not specify a direction
for the flow, the PME simply assumes that the first packet observed
for a flow is travelling forward, i.e. from source to destination.
In later analysis of the flow data, however, one is usually
interested in traffic to or from a particular source.

One can achieve this in a simple manner by writing a rule set to
specify the source for flows. All that is required is to have rules
which succeed if the packet is travelling in the required direction,
and which execute a 'Fail' action otherwise. This is demonstrated in
the following two examples.

(Note that early versions of NeMaC allowed 'Retry' as a synonym for
'Fail.' The current version also allows 'NoMatch,' which seems a
better way to imply "fail, allowing PME to try a second match with
directions reversed.")

```
  #  Count IP packets from network 130.216.0.0
  #
  SourcePeerType & 255 = IP: Pushto, ip_pkt;
  Null & 0 = 0: Ignore, 0;
  #
  ip_pkt:
    SourcePeerAddress & 255.255.0.0 = 130.216.0.0: Goto c_pkt;
    Null & 0 = 0: NoMatch, 0;
  #
  c_pkt:
    SourcePeerAddress & 255.255.255.255 = 0: PushPkttoAct, Next;
    DestPeerAddress & 255.255.255.255 = 0: CountPkt, 0;
```

The rule labelled ip_pkt tests whether the packet came from network
130.216. If it did not, the test fails and the following rule
executes a NoMatch action, causing the PME to retry the match with
the directions reversed. If the second match fails the packet did
not have 130.216 as an end-point, and is ignored.

The next rule set meters IP traffic on a network segment which
connects two routers, g1 and g2. It classifies flows into three
groups - those travelling from g1 to g2, those whose source is g1 and
those whose source is g2.

```
     #   Count IP packets between two gateways
     #
     #     -------+-----------------+-----------------+-------
     #         |                 |                 |
     #       +----+-----+       +----+-----+       +---+---+
     #       |  g1      |       |  g2      |       | meter |
     #       +-+-+-+-+--+       +-+-+-+-+--+       +-------+
     #
     SourcePeerType & 255 = IP: Pushto, ip_pkt;
     Null & 0 = 0: Ignore, 0;
     #
     ip_pkt:
       SourceAdjacentAddress & FF-FF-FF-FF-FF-FF = 00-80-48-81-0E-7C:
           Goto, s1;
       Null & 0 = 0: Goto, s2;
     s1:
       DestAdjacentAddress & FF-FF-FF-FF-FF-FF = 02-07-01-04-ED-4A
           GotoAct, g3;
       Null & 0 = 0: GotoAct, g1;
     s2:
       SourceAdjacentAddress & FF-FF-FF-FF-FF-FF = 02-07-01-04-ED-4A:
           Goto, s3;
       Null & 0 = 0: NoMatch, 0;
     s3:
       DestAdjacentAddress & FF-FF-FF-FF-FF-FF = 00-80-48-81-0E-7C:
           NoMatch, 0;
       Null & 0 = 0: GotoAct, g2;
     #
     g1: FlowClass & 255 = 1:  PushtoAct, c_pkt;  # From g1
     g2: FlowClass & 255 = 2:  PushtoAct, c_pkt;  # From g2
     g3: FlowClass & 255 = 3:  PushtoAct, c_pkt;  # g1 to g2
     #
     c_pkt:
       SourceAdjacentAddress & FF-FF-FF-FF-FF-FF = 0:
           PushPkttoAct, Next;
       DestAdjacentAddress & FF-FF-FF-FF-FF-FF = 0: PushPkttoAct, Next;
       SourcePeerAddress & 255.255.255.255 = 0: PushPkttoAct, Next;
       DestPeerAddress & 255.255.255.255 = 0: PushPkttoAct, Next;
       Null & 0 = 0:  Count, 0
```

   The first two rules ignore non-IP packets. The next two rules Goto
   s1 if the packet's source was g1, or to s2 otherwise. The rule
   labelled s2 tests whether the packet's source was g2; if not a
   NoMatch action is executed, allowing the PME to try the match with
   the packet's direction reversed. If the match fails on the second
   try the packet didn't come from (or go to) g1 or g2, and is ignored.

Packets which come from g1 are tested by the rule labelled s1, and
the PME will Goto either g3 or g1.

Packets which came from g2 are tested by the rule labelled s3. If
they are not going to g1 the PME will Goto g2. If they are going to
g1 a NoMatch action is executed - we want them counted as backward-
travelling packets for the g1-g2 flow.

The rules at g1, g2 and g3 push the value 1, 2 or 3 from their rule
into the flow's FlowClass attribute. This value can be used by an
Analysis Application to separate the flows into the three groups of
interest. FlowClass is an example of a 'computed' attribute, i.e.
one whose value is Pushed by the PME during rule matching.

The remaining rules Push the values of other attributes required for
later analysis, then Count the flow.

3.3 Subroutines

Subroutines are implemented in the PME in much the same way as in
BASIC. A subroutine body is just a sequence of statements, supported
by the GoSub and Return actions. 'GoSub' saves the PME's running
environment and jumps to the first rule of the subroutine body.
Subroutine calls may be nested as required - NeTraMet defines the
maximum nesting at compile time. 'Return n' restores the environment
and jumps to the action part of the nth rule after the Gosub, where n
is the index value from the Return rule.

The Return action provides a way of influencing the flow of control
in a rule set, rather like a FORTRAN Computed Goto. This is one way
in which a subroutine can return a result. The other way is by
Pushing a value in either a computed attribute (as demonstrated in
the preceding section), or in a flow attribute.

One common use for a subroutine is to test whether a packet attribute
matches one of a set of values. Such a subroutine becomes much more
useful if it can be used to test one of several attributes. The PME
architecture provides for this by using 'meter variables' to hold the
names of the attributes to be tested. The meter variables are called
V1, V2, V3, V4 and V5, and the Assign action is provided to set their
values. If, for example, we need a subroutine to test either
SourcePeerAddress or DestPeerAddress, we write its rules to test V1
instead. Before calling the subroutine we Assign SourcePeerAddress
to V1; later tests of V1 are converted by the PME into tests on
SourcePeerAddress. Note that since meter variables may be reassigned
in a subroutine, their values are part of the environment which must
be saved by a Gosub action.

The following rule set demonstrates the use of a subroutine ..

```
  #  Rule specification file to tally IP packets in three groups:
  #     UA to AIT, UA to elsewhere, AIT to elsewhere
  #
  #    -------+-----------------+----------------+--------
  #           |                 |                |
  #      +----+-----+      +----+-----+      +---+---+
  #      |   UA     |      |   AIT    |      | meter |
  #      +-+-+-+-+--+      +-+-+-+-+--+      +-------+
  #
    SourcePeerType & 255 = IP:      PushtoAct, ip_pkt;
    Null & 0 = 0:                   Ignore, 0;
  #
  ip_pkt:
    v1 & 0 = SourcePeerAddress:  AssignAct, Next;
    Null & 0 = 0:  Gosub, classify;
      Null & 0 = 0:  GotoAct, from_ua;     # 1 ua
      Null & 0 = 0:  GotoAct, from_ait;    # 2 ait
      Null & 0 = 0:  NoMatch, 0;           # 3 other
  #
  from_ua:
    v1 & 0 = DestPeerAddress:  AssignAct, Next;
    Null & 0 = 0:  Gosub, classify;
      Null & 0 = 0:  Ignore, 0;            # 1 ua-ua
      Null & 0 = 0:  GotoAct, ok_pkt;      # 2 ua-ait
      Null & 0 = 0:  Gotoact, ok_pkt;      # 3 ua-other
  #
  from_ait:
    v1 & 0 = DestPeerAddress:  AssignAct, Next;
    Null & 0 = 0:  Gosub, classify;
      Null & 0 = 0:  NoMatch, 0;           # 1 ait-ua
      Null & 0 = 0:  Ignore, 0;            # 2 ait-ait
      Null & 0 = 0:  GotoAct, ok_pkt;      # 3 ait-other
  #
  ok_pkt:
    Null & 0 = 0:                      Count, 0;
```

The subroutine begins at the rule labelled classify (shown below).
It returns to the first, second or third rule after the invoking
Gosub rule, depending on whether the tested PeerAddress is in the UA,
AIT, or 'other' group of networks. In the listing below only one
network is tested in each of the groups - it is trivial to add more
rules (one per network) into either of the first two groups. In this
example the subroutine Pushes the network number from the packet into
the tested attribute before returning.

The first invocation of classify (above) begins at the rule labelled
ip_pkt. It Assigns SourcePeerAddress to V1 then executes a Gosub
action. Classify returns to one of the three following rules. They
will Goto from_ua or from_ait if the packet came from the UA or AIT
groups, otherwise the PME will retry the match. This means that
matched flows will have a UA or AIT network as their source, and
flows between other networks will be ignored.

The next two invocations of 'classify' test the packet's
DestPeerAddress. Packets from AIT to UA are Retried, forcing them to
be counted as AU to AIT flows. Packets from UA to UA are ignored, as
are packets from AIT to AIT.

```
  classify:
    v1 & 255.255.0.0 = 130.216.0.0:  GotoAct, ua;   # ua
    v1 & 255.255.0.0 = 156.62.0.0:   GotoAct, ait;  # ait
    Null & 0 = 0:                     Return, 3;      # other
  ua:
    v1 & 255.255.0.0 = 0:            PushPkttoAct, Next;
    Null & 0 = 0:                    Return, 1;
  ait:
    v1 & 255.255.0.0 = 0:            PushPkttoAct, Next;
    Null & 0 = 0:                    Return, 2;
```

3.4 More complicated rule sets

The next example demonstrates a way of grouping IP flows together
depending on their Transport Address, i.e. their IP port number.
Simply Pushing every flow's SourceTransAddress and DestTransAddress
would produce a large number of flows, most of which differ only in
one of their transport addresses (the one which is not a well-known
port).

Instead we Push the well-known port number into each flow's
SourceTransAddress; its DestTransAddress will be zero by default.

```
    SourcePeerType & 255 = dummy:       Ignore, 0;
    SourcePeerType & 255 = IP:          Pushto, IP_pkt;
    Null & 0 = 0:   GotoAct, Next;
    SourcePeerType & 255 = 0:        PushPkttoAct, Next;
    Null & 0 = 0:  Count, 0;  # Count others by protocol type
  #
  IP_pkt:
    SourceTransType & 255 = tcp:     Pushto, tcp_udp;
    SourceTransType & 255 = udp:     Pushto, tcp_udp;
    SourceTransType & 255 = icmp:    CountPkt, 0;
    SourceTransType & 255 = ospf:    CountPkt, 0;
    Null & 0 = 0:  GotoAct, c_unknown;  #  Unknown transport type
```

```
  #
  tcp_udp:
  s_domain:
    SourceTransAddress & 255.255 = domain:  PushtoAct, c_well_known;
  s_ftp:
    SourceTransAddress & 255.255 = ftp:     PushtoAct, c_well_known;
  s_imap:
    SourceTransAddress & 255.255 = 113:     PushtoAct, c_well_known;
  s_nfs
    SourceTransAddress & 255.255 = 2049:    PushtoAct, c_well_known;
  s_pop:
    SourceTransAddress & 255.255 = 110:     PushtoAct, c_well_known;
  s_smtp:
    SourceTransAddress & 255.255 = smtp:    PushtoAct, c_well_known;
  s_telnet:
    SourceTransAddress & 255.255 = telnet:  PushtoAct, c_well_known;
  s_www:
    SourceTransAddress & 255.255 = www:     PushtoAct, c_well_known;
  s_xwin
    SourceTransAddress & 255.255 = 6000:    PushtoAct, c_well_known;
  #
    DestTransAddress & 255.255 = domain:    GotoAct, s_domain;
    DestTransAddress & 255.255 = ftp:       GotoAct, s_ftp;
    DestTransAddress & 255.255 = 113:       GotoAct, s_imap;
    DestTransAddress & 255.255 = 2049:      GotoAct, s_nfs;
    DestTransAddress & 255.255 = 110:       GotoAct, s_pop;
    DestTransAddress & 255.255 = smtp:      GotoAct, s_smtp;
    DestTransAddress & 255.255 = telnet:    GotoAct, s_telnet;
    DestTransAddress & 255.255 = www:       GotoAct, s_www;
    DestTransAddress & 255.255 = 6000:      GotoAct, s_xwin;
  #
    Null & 0 = 0:  GotoAct, c_unknown;  #  'Unusual' port
  #
  c_unknown:
    SourceTransType    & 255 = 0:      PushPkttoAct, Next;
    DestTransType      & 255 = 0:      PushPkttoAct, Next;
    SourceTransAddress & 255.255 = 0: PushPkttoAct, Next;
    DestTransAddress   & 255.255 = 0: CountPkt, 0;
  #
  c_well_known:
    Null & 0 = 0:  Count, 0
  #
```

   The first few rules ignore dummy packets, select IP packets for
   further processing, and count packets for other protocols in a single
   flow for each PeerType. TCP and UDP packets cause the PME to Push
   their TransType and Goto tcp_udp. ICMP and OSPF packets are counted
   in flows which have only their TransType Pushed.

At tcp_udp the packets' SourceTransAddress is tested to see whether
it is included in a set of 'interesting' port numbers. If it is, the
port number is pushed from the rule into the SourceTransAddress
attribute, and the packet is counted at c_well_known. (NeMaC accepts
Pushto as a synonym for PushRuleto).

This testing is repeated for the packet's DestTransAddress; if one of
these tests succeeds the PME Goes to the corresponding rule above and
Pushes the port number into the flow's SourceTransAddress. If these
tests fail the packet is counted at c_unknown, where all the flow's
Trans attributes are pushed. For production use more well-known
ports would need to be included in the tests above - c_unknown is
intended only for little-used exception flows!

Note that these rules only Push a value into a flow's
SourceTransAddress, and they don't contain any NoMatch actions. They
therefore don't specify a packet's direction, and they could be used
in other rule sets to group together flows for well-known ports.

The last example (below) meters flows from a remote router, and
demonstrates another approach to grouping well-known ports.

```
  SourceAdjacentAddress & FF-FF-FF-FF-FF-FF =
      00-60-3E-10-E0-A1:  Goto, gateway;  # tmkr2 router
  DestAdjacentAddress & FF-FF-FF-FF-FF-FF = 00-60-3E-10-E0-A1:
      Goto, gateway;  # Source is tmkr2
  Null & 0 = 0:  Ignore, 0;
#
gateway:
  SourcePeerType & 255 = IP:  GotoAct, IP_pkt;
  Null & 0 = 0:  GotoAct, Next;
  SourcePeerType & 255 = 0:  CountPkt, 0;
#
IP_pkt:
  SourceTransType & 255 = tcp:    PushRuleto, tcp_udp;
  SourceTransType & 255 = udp:    PushRuleto, tcp_udp;
  Null & 0 = 0: GotoAct, not_wkp;  #  Unknown transport type
#
tcp_udp:
  SourceTransAddress & FC-00 = 0:  GotoAct, well_known_port;
  DestTransAddress & FC-00 = 0:  NoMatch, 0;
  Null & 0 = 0:  GotoAct, not_wkp;
#
not_wkp:
  DestTransAddress    & 255.255        = 0: PushPkttoAct, Next;
well_known_port:
  SourcePeerType      & 255            = 0: PushPkttoAct, Next;
  DestPeerType        & 255            = 0: PushPkttoAct, Next;
```

```
   SourcePeerAddress   & 255.255.255.0 = 0: PushPkttoAct, Next;
   DestPeerAddress     & 255.255.255.0 = 0: PushPkttoAct, Next;
   SourceTransType     & 255           = 0: PushPkttoAct, Next;
   DestTransType       & 255           = 0: PushPkttoAct, Next;
   SourceTransAddress & 255.255        = 0: CountPkt, 0;
```

The first group of rules test incoming packet's AdjacentAddresses to
see whether they belong to a flow with an end point at the specified
router. Any which don't are ignored. Non-IP packets are counted in
flows which only have their PeerType Pushed; these will produce one
flow for each non-IP protocol. IP packets with TransTypes other than
UDP and TCP are counted at not_wkp, where all their address
attributes are pushed.

The high-order six bits of SourceTransAddress for UDP and TCP packets
are compared with zero. If this succeeds their source port number is
less than 1024, so they are from a well-known port. The port number
is pushed from the rule into the flow's SourceTransAddress attribute,
and the packet is counted at well_known_port. If the test fails, it
is repeated on the packet's DestTransAddress. If the destination is
a well-known port the match is Retried, and will succeed with the
well-known port as the flow's source.

If later analysis were to show that a high proportion of the observed
flows were from non-well-known ports, further pairs of rules could be
added to perform a test in each direction for other heavily-used
ports.

4 Flow data files

Although the Architecture document [1] specifies - in great detail -
how the Traffic Flow Meter works, and how a meter reader should
collect flow data from a meter, it does not say anything about how
the collected data should be stored. NeMaC uses a simple, self-
documenting file format, which has proved to be very effective in
use.

There are two kinds of records in a flow data file: flow records and
information records. Each flow record is simply a sequence of
attribute values with separators (these can be specified in a NeMaC
rule file) or spaces between them, terminated by a newline.

Information records all start with a cross-hatch. The file's first
record begins with ##, and identifies the file as being a file of
data from NeTraMet. It records NeMaC's parameters and the time this
collection was started. The file's second record begins with
#Format: and is a copy of the Format statement used by NeMaC to
collect the data.

   The rest of the file is a sequence of collected data sets. Each of
   these starts with a #Time: record, giving the time-of-day the
   collection was started, the meter name, and the range of meter times
   this collection represents. These from and to times are meter
   UpTimes, i.e. they are times in hundredths of seconds since the meter
   commenced operation. Most analysis applications have simply used the
   collection start times (which are ASCII time-of-day values), but the
   from and to times could be used to convert Uptime values to time-of-
   day. The flow records which comprise a data set follow the #Time
   record.

4.1 Sample flow data file

   A sample flow data file appears below. Most of the flow records have
   been deleted, but lines of dots show where they were.

   ##NeTraMet v3.2. -c300 -r rules.lan -e rules.default
     test_meter -i eth0 4000 flows starting at 12:31:27 Wed 1 Feb 95
   #Format: flowruleset flowindex firsttime sourcepeertype
     sourcepeeraddress destpeeraddress topdus frompdus
     tooctets fromoctets
   #Time: 12:31:27 Wed 1 Feb 95 130.216.14.251 Flows
     from 1 to 3642
   1 2 13   5 31.32.0.0 33.34.0.0 1138 0   121824 0
   1 3 13   2 11.12.0.0 13.14.0.0 4215 0   689711 0
   1 4 13   7 41.42.0.0 43.34.0.0 1432 0   411712 0
   1 5 13   6 21.22.0.0 23.24.0.0 8243 0   4338744 0
   3 6 3560   2 130.216.14.0 130.216.3.0 0 10   0 1053
   3 7 3560   2 130.216.14.0 130.216.76.0 59 65   4286 3796
   3 8 3560   7 0.0.255.0 1.144.200.0 0 4   0 222
   3 9 3560   2 130.216.14.0 130.216.14.0 118 1   32060 60
   3 10 3560   6 130.216.0.28 130.216.0.192   782 1   344620 66
   3 11 3560   7 0.0.255.0 0.128.113.0   0 1   0 73
   3 12 3560   5 59.3.13.0 4.1.152.0   1 1   60 60
   3 13 3560   7 0.128.94.0 0.129.27.0   2 2   120 158
   3 14 3560   5 59.3.40.0 4.1.153.0   2 2   120 120
   3 15 3560   5 0.0.0.0 4.1.153.0   0 1   0 60
   3 16 3560   5 4.1.152.0 59.2.189.0   2 2   120 120

        . .   .   .
   3 42 3560   7 0.128.42.0 0.129.34.0   0 1   0 60
   3 43 3560   7 0.128.42.0 0.128.43.0   0 1   0 60
   3 44 3560   7 0.128.42.0 0.128.41.0   0 1   0 60
   3 45 3560   7 0.128.42.0 0.129.2.0   0 1   0 60
   3 46 3560   5 4.1.152.0 59.2.208.0   2 2   120 120
   3 47 3560   5 59.3.46.0 4.1.150.0   2 2   120 120
   3 48 3560   5 4.1.152.0 59.2.198.0   2 2   120 120
   3 49 3560   5 0.0.0.0 59.2.120.0   0 1   0 60
   3 50 3664   5 4.1.152.0 59.2.214.0   0 1   0 60

```
 3 51 3664   5 0.0.0.0 4.2.142.0   0 1   0 60
 3 52 3664   5 4.1.153.0 59.3.45.0   4 4   240 240
#Time: 12:36:25 Wed  1 Feb 95 130.216.14.251 Flows
  from 3641 to 33420
 3 6 3560   2 130.216.14.0 130.216.3.0   0 21   0 2378
 3 7 3560   2 130.216.14.0 130.216.76.0   9586 7148   1111118 565274
 3 8 3560   7 0.0.255.0 1.144.200.0   0 26   0 1983
 3 9 3560   2 130.216.14.0 130.216.14.0   10596 1   2792846 60
 3 10 3560   6 130.216.0.28 130.216.0.192   16589 1   7878902 66
 3 11 3560   7 0.0.255.0 0.128.113.0   0 87   0 16848
 3 12 3560   5 59.3.13.0 4.1.152.0   20 20   1200 1200
 3 13 3560   7 0.128.94.0 0.129.27.0   15 14   900 1144
 3 14 3560   5 59.3.40.0 4.1.153.0   38 38   2280 2280
 3 15 3560   5 0.0.0.0 4.1.153.0   0 30   0 1800
 3 16 3560   5 4.1.152.0 59.2.189.0   20 20   1200 1200
 3 17 3560   5 0.0.0.0 59.2.141.0   0 11   0 660

     .   .   .   .   .   .   .   .
 3 476 26162   7 0.129.113.0 0.128.37.0   0 1   0 82
 3 477 27628   7 0.128.41.0 0.128.46.0   1 1   543 543
 3 478 27732   7 0.128.211.0 0.128.46.0   1 1   543 543
 3 479 31048   7 0.128.47.0 2.38.221.0   1 1   60 60
 3 480 32717   2 202.14.100.0 130.216.76.0   0 4   0 240
 3 481 32717   2 130.216.76.0 130.216.3.0   0 232   0 16240
#Time: 12:41:25 Wed  1 Feb 95 130.216.14.251 Flows
  from 33419 to 63384
 3 6 3560   2 130.216.14.0 130.216.3.0   51 180   3079 138195
 3 7 3560   2 130.216.14.0 130.216.76.0   21842 18428   2467693 1356570
 3 8 3560   7 0.0.255.0 1.144.200.0   0 30   0 2282
 3 9 3560   2 130.216.14.0 130.216.14.0   24980 1   5051834 60
 3 10 3560   6 130.216.0.28 130.216.0.192   20087 1   8800070 66
 3 11 3560   7 0.0.255.0 0.128.113.0   0 164   0 32608
 3 12 3560   5 59.3.13.0 4.1.152.0   41 41   2460 2460
 3 14 3560   5 59.3.40.0 4.1.153.0   82 82   4920 4920
 3 15 3560   5 0.0.0.0 4.1.153.0   0 60   0 3600

     .   .   .   .   .   .   .   .
```

4.2 Flow data file features

  Several features of NeMaC's flow data files (as indicated above) are
  worthy of note:

  - Collection times overlap slightly between samples. This allows for
    flows which were created after the collection started, and makes
    sure that flows are not missed from a collection.

   - The rule set may change during a run. The above shows flows from
     rule set 1 - the default set - in the first collection, followed by
     the first flows created by rule set 3 (which has just been
     downloaded by NeMaC).

   - FlowIndexes may be reused by the meter once their flows have been
     recovered by the garbage collector. The combination of
     FlowRuleSet, FlowIndex and StartTime are needed to identify a flow
     uniquely.

   - Packet and Byte counters are 32-bit unsigned integers, and are
     never reset by the meter. Computing the counts occurring within a
     collection interval requires taking the difference between the
     collected count and its value when the flow was last collected.
     Note that counter wrap-around can be allowed for by simply
     performing an unsigned subtraction and ignoring any carry.

   - In the sample flow data file above I have used double spaces as
     separators between the flow identifiers, peer addresses, pdu counts
     and packet counts.

   - The format of addresses in the flow data file depends on the type
     of address. NeMaC always displays Adjacent addresses as six hex
     bytes separated by hyphens, and Transport addresses as (16-bit)
     integers. The format of a Peer address depends on its PeerType,
     e.g. dotted decimal for IP. To facilitate this NeMaC needs to know
     the PeerType for each flow; the user must request NeMaC to collect
     it.

4.3 Terminating and restarting meter reading

   When NeMaC first starts collecting from a meter, it reads the flow
   data for all active flows. This provides a starting point for
   analysis applications to compute the counts between successive
   collections.

   From time to time the user needs to terminate a flow data file and
   begin a new one. For example, a user might need to generate a
   separate file for each day of metering. NeMaC provides for this by
   closing the file after each collection, then opening it and appending
   the data from the next collection. To terminate a file the user
   simply renames it. The Unix system will effect the name change
   either immediately (if the file was closed) or as soon as the current
   collection is complete (and the file is closed).

   When NeMaC begins its next collection it observes that the file has
   disappeared, so it creates a new one and writes the # header records
   before writing the collected data.

There is one aspect of the above which requires some care on the
user's part. The last data set in a file is not duplicated as the
first data set of the next file. In other words, analysis
applications must either look ahead at the first data set of the next
file, or begin by reading the last data set of the previous file. If
they fail to do this they will loose one collection's worth of flow
data at each change of file.

5 Analysis applications

   Most analysis applications will be unique, taking data produced by a
   locally-developed rule set and producing reports to satisfy specific
   local requirements. The NeTraMet distribution files include three
   applications which are of general use, as follows:

   - fd_filter computes data rates, i.e. the differences between
     successive data sets in a flow data file. It also allows the user
     to assign a 'tag' number to each flow; these are 'computed'
     attributes similar to FlowClass and FlowKind - the only difference
     is that they are computed from the collected data sets.

   - fd_extract takes 'tagged' files from fd_filter and produces simple
     'column list' files for use by other programs. One common use for
     fd_extract is to produce time-series data files which can be plotted
     by utilities like GNUPlot.

   - nm_rc is a 'remote console' for a NeTraMet meter. It is a slightly
     simplified version of NeMaC combined with fd_filter. It can be used
     to monitor any meter, and will display (as lines of text
     characters) information about the n busiest flows observed during
     each collection interval.

   - nifty is a traffic flow analyser, which (like nm_rc) displays data
     from a NeTraMet meter. nifty is an X/Motif application, which
     produces displays like 'Packet rate (pps) vs Flow lifetime
     (minutes),' so as to highlight those flows which are 'interesting.'

   These applications are useful in themselves, and they provide a good
   starting point for users who wish to write their own analysis
   applications.

6 Using NeTraMet in a measurement system

   This section gives a brief summary of the steps involved in setting
   up a traffic measurement system using NeTraMet. These are:

  - Decide what is to be measured. One good way to approach this is to
    specify exactly which flows are to be measured, and what reports
    will be required. Specifying the flows should make it obvious
    where meters will have to be placed so that the flows can be
    observed, whether PCs will be adequate for the task, etc..

  - Install meters. As well as actually placing the meter hosts this
    includes making sure that they are configured correctly, with
    appropriate IP addresses, SNMP community strings, etc.

  - Develop the rule set (and a standby rule set). The degree of
    difficulty here depends on how much is known in advance about the
    traffic. One possible approach is to start with the meter default
    rule set and measure how much traffic there is for each PeerType.
    (This is a good way to verify that NeTraMet and NeMaC are working
    properly). You can now add rules so as to increase the granularity
    of the flows; this will of course increase the number of flows to
    be collected, and force the meter's garbage collector to work
    harder. Another approach is to try a rule set with very fine
    granularity (i.e. one which Pushes all the address attributes),
    then observing how many flows are collected every few minutes.

  - Develop a strategy for controlling meter reader. This means
    setting the meter's maximum number of flows, the collection
    interval, how breaks between flow data files will be handled, how
    often NeMaC should check that the meter is running, etc.

  - Develop application(s) to process the collected flow data and
    produce the required files and reports.

  - Test run. Monitor the system, then refine the rule sets and meter
    reading strategy until the overall system performance is
    satisfactory.

   This process can take quite a long time, but the overall result is
   well worth the effort.

6.1 Examples of NeTraMet in production use

   At the University of Auckland we run two sets of meters. One of
   these measures the traffic entering and leaving our University
   network, and generates usage reports for all our Internet users.
   This has been in production since early 1994.

The other set consists of meters which are distributed at
Universities throughout New Zealand. They provide continuous traffic
flow measurements at five-minute intervals for all the links making
up the Universities' network (Kawaihiko); this system has been in
production since January 1996, and has already proved very useful in
planning the network's development.

The Kawaihiko Network provides IP connectivity for the New Zealand
Universities. They are linked via a Frame Relay cloud, using a
partial mesh of permanent virtual circuits. There is a NeTraMet
meter at each site, metering inward and outward traffic. All the
meters are managed from Auckland, and they all run copies of the same
rule set.

The rule set has about 650 rules, most of which are in a single
subroutine which classifies PeerAddresses into three categories -
'Kawaihiko network,' 'other New Zealand network' and 'non-New Zealand
network.' Inside New Zealand IP addresses lie within six CIDR
blocks, and there are about four hundred older networks which have
addresses outside those blocks. The rules are arranged in groups by
subnet size, i.e. all the /24 networks are tested first, then the /23
networks, etc, finishing with the /16 networks. This means that
although there are about 600 networks, any PeerAddress can be
classified with only nine tests.

The Kawaihiko rule set classifies flows, using computed attributes to
indicate the network 'kind' (Kawaihiko / New Zealand / international)
for each flow's SourcePeerAddress and DestPeerAddress, and to
indicate whether the flow is a 'network news' flow or not.

Flow data is collected from all of the meters every five minutes, and
is used to produce weekly reports, as follows:

- Traffic Plots. Plots of the 5-minute traffic rates for each site,
  showing international traffic in and out, news traffic in and out,
  and total traffic in and out of the site.

- Traffic Matrices. Two of these are produced, one for news traffic,
  the other for total traffic. They show the traffic rates from
  every site (including 'other New Zealand' and 'international') to
  every other site. The mean, third quartile and maximum are printed
  for every cell in the matrices.

7 Acknowledgments

This memo documents the implementation work on traffic flow
measurement here at the University of Auckland. Many of my
University colleagues have contributed significantly to this work,
especially Russell Fulton (who developed the rules sets, Perl scripts
and Cron jobs which produce our traffic usage reports automatically
week after week) and John White (for his patient help in documenting
the project).

8 References

[1] Brownlee, N., Mills, C., and G. Ruth, "Traffic Flow
Measurement: Architecture", RFC 2063, The University of Auckland,
Bolt Beranek and Newman Inc., GTE Laboratories, Inc, January 1997.

[2] Brownlee, N., "Traffic Flow Measurement: Meter MIB",
RFC 2064, The University of Auckland, January 1997.

[3] CRYNWR Packer Drivers distribution site:
http://www.crynwr.com/

[4] Case J., McCloghrie K., Rose M., and Waldbusser S.,
"Structure of Management Information for version 2 of the
Simple Network Managemenet Protocol", RFC 1902, SNMP Research
Inc., Hughes LAN Systems, Dover Beach Consulting, Carnegie
Mellon University, April 1993.

[5] IBM Corporation, "IBM PC Technical Reference Manual," 1984.

[6] Waterloo TCP distribution site:
http://mvmpc9.ciw.uni-karlsruhe.de:80/d:/public/tcp_ip/wattcp

[7] CMU SNMP distribution site:
ftp://lancaster.andrew.cmu.edu/pub/snmp-dist

[8] libpcap distribution site:
ftp://ftp.ee.lbl.gov/libpcap-*.tar.gz

9 Security Considerations

   Security issues are not discussed in detail in this document. The
   meter's management and collection protocols are responsible for
   providing sufficient data integrity and confidentiality.

10 Author's Address

   Nevil Brownlee
   The University of Auckland

   Phone: +64 9 373 7599 x8941
   Email: n.brownlee@auckland.ac.nz