REFERENCE MANUAL

NeTraMet & NeMaC

Network Traffic Meter & NeTraMet Manager/Collector

Version 4.3

Nevil Brownlee

Information Technology Systems & Services
The University of Auckland
Auckland, New Zealand

June 1999

PREFACE

It is a pleasure to write this preface to Nevil Brownlee's introductory documentation for NeTraMet and NeMaC. The system, which collects accounting data for network traffic, has already proved to be a gold-mine (in the figurative sense), providing information about network traffic flows and activity patterns which are invaluable in reaching a better understanding of network usage.

Dr John C. B. White
Director, Computer Centre
University of Auckland
October 199

CONTENTS

Nevil Brownlee is the Director, Technology Development, of The University of Auckland's Information Technology Systems & Services group, and is responsible for support and development of the University's campus network, which has about 6,000 connected hosts. He co-ordinates Kawaihiko, the New Zealand Universities' network, and is active within the IETF, especially on the Realtime Traffic Flow Measurement (RTFM) Working Group. He holds a Ph.D. degree in Atmospheric Physics.

1. Introduction

NeTraMet is a meter for network traffic flows (see below for definitions), and is the first implementation of the Realtime Traffic Flow Measurement (RTFM) Working Group's Measurement Architecture. This is outlined in RFC 1272, "Internet Accounting Background," and has three components:

- meters, i.e. small hosts which are attached to a network segment and measure traffic flowing on that segment;
- meter readers, which retrieve information from meters;
- managers, which instruct meters as to which flows they should measure and meter readers as to which meters they should collect from, at what intervals.

A meter reader can collect flow data from many meters, and each meter may have its data retrieved by several meter readers. Traffic flows of interest are defined by user-specified sets of rules.

1.1. Operating Environments

This release of NeTraMet runs on a Solaris, SunOS, Irix or Linux host, using libpcap to observe Ethernet packet headers, or on a PC using a CRYNWYR packet driver. The PC implementation can be used with a 25 MHz 386SX, where it will handle about 1250 packets per second, and can cope with peak traffic bursts of up to 2250 packets per second for several seconds at a time. On a 40 MHz 486 NeTraMet will handle 3000 pps peaks. On a 60 MHz Pentium with a PCI-bus card NeTraMet can easily handle a steady load of 6,000 packets per second.

This release also includes NeMaC (NeTraMet Manager/Collector), a combined manager and collector program. It runs on Unix systems, and is in use on Solaris, SunOS, Irix, Linux, DEC Unix, AIX and HPUX systems.

GNU autoconfig is used to build Makefiles for all the NeTraMet programs, which makes it very straightforward to create and install the NeTraMet programs on a Unix system.

1.2. Traffic Flows

A traffic flow is a stream of packets exchanged between two network hosts, which we refer to as the flow's source and destination. Flows are bi-directional in that packets and bytes can be counted in the 'to' (source to destination) and 'from' (destination to source) directions.

The 'identity' of a flow is determined by the address attributes of its two hosts, and these attributes can be of three kinds:

adjacent	(link layer)
peer	(network layer)
transport	(transport layer)

When NeTraMet is being used to meter on an Ethernet, *adjacent addresses* will be Ethernet addresses.

A *peer address* can be an IP address, a DECnet phase IV address, a Novell network number, an EtherTalk address or a CLNS NSAP, these being the five protocols currently understood by NeTraMet.

A *transport address* contains specifications for details within the peer protocol. For IP these are the protocol type and source and destination port numbers, and similar kinds of detail are defined for the other protocols.

Within the meter a flow is implemented as a data structure containing the attributes of its source and destination, its packet and byte counters, the times it was first and last observed, and other information used for control purposes.

The meter could simply create flows for every possible combination of source and destination attributes it observes, but this would quickly exhaust its memory. Instead the meter uses a set of rules to decide which flows are of interest, and other packets are ignored.

Each rule tests one attribute of a flow, using a mask to specify which bits are of interest. In this way a tree of rules can be built up to classify packets into flows; each packet can then be 'counted' in its appropriate flow. If this is all that the rules specify, no further information about the flow is retained in the meter.

If more detail is required, the rules can instruct the meter to 'tally' the packet, i.e, create many sub-flows instead of a single flow. For example they might determine that a packet has come from a class B IP source, then tally it into flows for each of the source network's class C subnets. Tallying in this style is implemented by using rules to extract information from the packet using wider masks than those which were used in earlier rules - examples of rules files which do this are given below.

NeTraMet can also count packets and bytes for protocols it does not understand. All such packets are effectively aggregated together in a single flow, which has the *peer type* Other.

1.3. The Traffic Flow Meter MIB

The Internet Accounting Group produced an Internet Draft describing its proposed Accounting Meter Services MIB, which used the number Experimental.99. This MIB, now known as the RTFM Traffic Meter MIB, has been assigned a proper Object Identifier by IANA; it is mib-2 40.

The draft was discussed over several IETF meetings until the Working Group became dormant in March 1993, and informally at meetings and via the Group's mailing list since then. A copy of the current version – which NeTraMet implements – is included with this release. It defines all of the MIB variables mentioned in this documentation.

The Realtime Traffic Flow Measurement Working Group has produced RFC 2064, 'Traffic Flow Measurement: Meter MIB,' which has since been updated in further Internet Drafts. This improved version of the Traffic Flow Meter MIB is implemented in version 4.1 of NeTraMet.

2. Implementation Details

2.1. Program Development Environment

NeTraMet was developed on a PC using Borland's Turbo C, Turbo Assembler and interactive development environment. Turbo Make was used to organise the system, with Make files specifying how the various system components are created.

Waterloo C provided a good PC implementation of TCP/IP, and was used to provide UDP transport for SNMP packets. Waterloo C interfaces with Ethernet via a CRYNWR packet driver. I extended the packet interface to support monitoring by running the Ethernet card in promiscuous mode and copying packet headers into a rotating buffer.

An early version of CMU SNMP was used for communication between NeTraMet and NeMaC. This was ported to the PC, and extended to support SET operations on character and (16-bit) integer variables.

From late 1995 I began work on using SNMPv2C instead of SNMP, which involved taking CMU SNMPv2 and removing the 'SNMP Security' extensions from it so as to leave only community-based security.

At about the same time I was able to port NeTraMet to run within OC3MON, an ATM traffic monitor system developed by NLANR and MCI. This involved moving to Borland C++ version 4.5 and Borland's DOS PowerPack 32-bit environment. As well as the OC3MON version of NeTraMet, this has allowed me to produce a 32-bit DOS version of the meter.

2.2. Data Structures: Flow and Rule Tables

Host addresses (adjacent, peer and transport) and their masks are held in a structure called a *key.* A flow is a larger structure which contains two keys, one each for source and destination host. General attributes are stored as variables within a flow, and there is a *link* field to enable flows to be linked together. Space for flows is allocated dynamically from a pool of flows. The flow table is implemented as an array of pointers to flows; a *FlowIndex* is an index into this array.

Rules are implemented in a smaller structure. Space for the rule table is allocated as a single block of memory.

The first time a flow is observed a *count* action is executed. The meter allocates space for the flow, assigns it a *FlowIndex,* and enters it in a *count table,* which is implemented as a single large hash table. The meter maintains a table of pointers to the count table; these are the primary means of accessing flow data.

2.3. Meter Memory Management

Once a flow has been created it could continue to exist indefinitely. In time, however, the meter will run out of space for new flows. To deal with this problem NeTraMet uses an incremental garbage collector.

At regular intervals specified by the *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 the parameters described in the 'PC Statistics display' section below.

To decide whether a flow can be recovered, the garbage collector considers how long it has been inactive (no packets in either direction), and when its data was last collected. If it has been collected at least once since its *LastTime,* it may be recovered. This alogrithm is implemented using a variable called *GarbageCollectTime,* which normally contains the meter's *UpTime* when the collection before last was started.

Should flows not be collected often enough the meter could run out of space. This is prevented by having a low-priority background process check the percentage of flows active and compare it with the *HighWaterMark* variable. If the percentage of active flows is greater than the high-water mark, *GarbageCollectTime* is incremented by the current value of the *InactivityTimeout* variable. The effect of this is that if a collector fails NeTraMet will continue to create flows until *HighWaterMark* is exceeded, then recover the oldest flows to maintain sufficient free memory.

The MIB specifies that a meter should switch to using an 'standby' rule table if the percentage of flows active rises above *HighWaterMark.* The MIB also specifies that the meter should take some action when the active flow percentage rises above *FloodMark;*

NeTraMet switches to the default rule set in this case. The values of the memory management variables mentioned above can all be set by NeMaC – it should be possible to tune them to work effectively in nearly all cases.

2.4. Optimising the Rule Table

Rules are commonly tested in sequence until a successful match selects a new index into the rule table. Testing a packet against a long list of addresses is thus a sequential search, which would be slow. To improve rule testing performance NeTraMet performs flow analysis of a new rule table, looking for groups of rules which test the same attribute using the same mask. Groups which have more than four rules are organised into small hash tables – these can effectively be tested with a hash computation and a single compare.

When a packet arrives at the meter its attributes are copied into the next available slot of the meter's rotating input buffer, ready for later matching against the rules. If the packet's *peer type* can never be matched by the rules, it is simply discarded (before its other attributes are copied). To implement this the meter maintains a table indicating which *peer types* are tested by the current rule set; this table can be displayed on the meter's screen at any time. Similarly, if the current rule set doesn't require *adjacent addresses* they are not copied into the input buffer.

2.5. The Meter's Outer Block

MeTraMet's outer block is a single loop which implements four asynchronous processes. These are – in decreasing priority order –

- *Handle SNMP requests*. Process these and send SNMP responses.
- *Monitor Ethernet packets* (including NeTraMet SNMP requests and responses). Test each against the active rule set(s) and count as required.
- *Handle keyboard commands.* Test the keyboard once each second, and process any incoming keystrokes.
- *Memory Management*. Attempt to recover memory, as described above. *GarbageCollectInterval* is set to 5 seconds by default, and *HighWaterMark* is tested every 30 seconds.

2.6. Matching the Rules

When a packet arrives at the meter two key data structures are built, one for its source and one for its destination. An attempt is made to match the packet against the current rule set with the keys in source-to-destination order; if this succeeds the packet is counted.

If the match fails, the keys are interchanged and the packet is tested against the rules again. If it fails this time it is discarded.

A third possibility is that the packet matched a count rule, but its flow was not yet present in the count table. Since it might already have been seen travelling in the opposite direction the match is retried with the keys interchanged. If this fails the flow is added into the count table with its keys in source-to-destination order.

This algorithm means that you can use symmetrical counts (i.e. counts having identical masks in both directions) if you don't care about flow direction. Alternatively you may write rules which enforce a particular source-to-destination order. Examples of these are given in the section on writing rule files below.

2.7. SNMP: Getting and Setting Variables

The code for getting and setting SNMP variables was developed from CMU's *snmpvars.c* file. I have modified this to use a binary search to find object identifiers as required, replacing the original linear search.

Simple procedures for setting character, (16-bit) integer and (32-bit) long variables are used when no special action is required. Special actions, e.g. updating *GarbageCollectTime* when the *LastCollectTime* variable is set, are implemented as individual procedures.

From version 4.1, 64-bit counters (64-bit unsigned) are used for packet and byte counters. This allows for higher-speed network links, and (more commonly) allows the meter to run for longer periods without counter roll-over.

2.8. Supporting Multiple Meter Readers

A rather unusual requirement for the accounting meter is the ability to support asynchronous collection of flow data by many meters simultaneously. This is described in the MIB by using rule set number and last-active time filter, as indexes to the flow table, allowing it to inspect only those flows belonging to a specified rule set and created or active since a specified time. This time is passed to the meter as one component of an object identifier; you can view this as a parameter being passed to the GET procedure which implements the creation or activity table.

Another aspect of having multiple collectors is that one of them may collect performance statistics. One possibility would be to have one meter collecting flow data every 15 minutes, a second collecting flow data every hour, and a third collecting performance statistics (but not flow data) every minute.

2.9. Recovering Bulk Flow Data

One vital element of an accounting meter is that it must be possible to retrieve flow data in an efficient manner. SNMP can be inefficient for this purpose, since every value retrieved is accompanied by its object identifier. To retrieve a long value (four bytes) can require a further 12 or more bytes of object identifier!

NeTraMet versions 2 and 3 solved this problem by using SNMP opaque objects to pass many values back to NeMaC as a single unit. The MIB defines an object called a *Column Blob* to do this. A *Column Blob* is a three-dimensional SNMP object, the dimensions being a 'column' number, a *LastTime* value and a *FlowIndex*. NeMaC views the flow table as a matrix with a column for each flow attribute. It can retrieve values of a particular attribute for all flows active since a specified time, starting at a given row of the flow table and recalling column blobs in sequence down a column.

NeMaC 2 & 3 took this idea a little further. The user specifies which attributes are to be collected using a Format statement in the rule file. NeMaC used the Format to decide which columns are required, then retrieved column blobs for each attribute starting at the first row of the flow table. The resulting collected flows were written to disk, then the process was repeated starting at the row after the last collected row, and so on.

The maximum column blob size was chosen to fit into a 500-byte SNMP packet, which can carry from 50 to 60 attribute values (together with their flow numbers). As an example, if we wished to collect 10 attributes for 1000 flows, this required only 10 x 20 = 200 SNMP packets. To minimise network loading, NeMaC pauses for a few milliseconds after each SNMP request.

With version 4.1, the new RFC 2064 Meter MIB introduced the notion of a 'flow data package.' This is a list of attribute values for a flow, retrieved from the meter via a single

SNMP GET request. Using SNMPv2's GETBULK request, a series of packages can be retrieved within a single (1500-byte) SNMP packet. This provides a data retrieval method with about the same level of overhead as the older column blobs, except that using 1500-byte packets rather than 500-byte ones reduces the number of packets needed. More important, all the attribute values for a flow are retrieved at the same time, which was not the case using column blobs. This provides a significant improvement for programs such as nifty, which need consistent data for each flow.

3. Flow Attributes

3.1 Introduction

A flow's attributes may be conveniently arranged into five groups: adjacent address, peer address, transport address, subscriber and general. Since NeTraMet can't determine subscriber information merely by watching the packets passing by, subscriber attributes are not currently implemented. A meter running in a network access server would, however, be able to implement them.

Adjacent address attributes are described in the Adjacent Attributes section; they are the same for all of the peer address types.

Peer and transport addresses, however, are different for each of the peer protocols. They are therefore explained together in the sections on IP, DECnet, Novell IPX and EtherTalk. The attributes which give a flow's peer and transport address type are *SourcePeerType* and *SourceTransType*. *SourcePeerType* and *DestPeerType* are synonyms, as are *SourceTransType* and *DestTransType*. If the meter were implemented on a gateway the source- and dest- types could be different, but this is impossible on a single network segment.

3.2. Adjacent attributes

NeTraMet's initial implementation only supports passive interfaces. This means that each flow has the same adjacent type for its two directions, i.e. *SourceAdjacentType* and *DestAdjacentType* have the same value.

The *SourceAdjacentAddress* and *DestAdjacentAddress* attributes are the Ethernet MAC addresses of the source and destination hosts. These are written as six hexadecimal bytes separated by hyphens, e.g. 00-00-C0-00-13-A5. They may be entered in this form, or as six decimal bytes separated by dots.

SourceAdjacentMask and *DestAdjacentMask* may be used in rules to test fields within adjacent addresses. They are written and entered in the same form as adjacent addresses.

3.3. IP attributes

SourcePeerType and *DestPeerType*   = 1

> IP in this context means IP version 4, which nm_rc displays as IP4. IP version 6 was implemented in version 4.3 of NeTraMet (to enable this one must make NeTraMet using the V6 compile-time option); its PeerType value is 2.

SourcePeerAddress and *stPeerAddress*

> IP addresses of the flow's two hosts, written as four decimal bytes separated by dots and entered in the same way, e.g. 130.216.234.237.

SourcePeerMask and *DestPeerMask*

Address masks for tests; in the same form as peer addresses.

DSCodePoint

The Differentiated Service Code Point (0..63) from the IP packet header. Note that because the header carries only a single code point a flow can't have separate Source- and Dest- CodePoint Attributes.

SourceTransType and *DestTransType*

Protocol field from the IP packet header. The values of these are given in RFC 1700, Assigned Numbers. Common values are:

1 =	ICMP	17 =	UDP
6 =	TCP	18 =	OSPF

These names can be used for them in rule files.

If the *TransType* is **TCP** or **UDP**, the *SourceTransAddress* and *DestTransAddress* contain the flow's source and destination port numbers. Many of their values are given in RFC 1700, Assigned Numbers. Common values are

20 =	FTP-DATA	70 =	GOPHER
21 =	FTP	80 =	WWW
23 =	TELNET	119 =	NNTP
25 =	SMTP	123 =	NTP
53 =	DOMAIN	161 =	SNMP

These names can be also be used for them in rule files

NeTraMet copes with fragmented IP packets as follows. All fragments contain peeraddresses and transport protocol type, so these attributes are handled normally. The first fragment of a packet is assumed to contain a complete UDP or TCP header, so transport addresses (IP port numbers) should be set correctly. Fragments other than the first don't have the header information, so their transport addresses are set to zero.

If the *TransType* is **ICMP**, then *SourceTransAddress* contains the ICMP type and *DestTransAddress* contains the ICMP code. For a description of these see Comer, "Internetworking with TCP/IP." Common values are:

0 =	echo reply	5 =	redirect
3 =	destination unreachable	8 =	echo request

SourceTransMask and *DestTransMask*

Like *SourceTransType*, these are 16-bit fields, which are written as a single integer, and can be entered in this form or as two decimal bytes separated by a dot, e.g. 255.255 or 65535.

3.4. DECnet attributes

SourcePeerType and *DestPeerType* = 13

SourcePeerAddress and *DestPeerAddress*

DECnet Phase IV addresses of the flow's two hosts, written as four decimal bytes separated by dots and entered in the same way, e.g.4.1.150.0. The first byte is the DECnet Area Number, the next two are the (16-bit) DECnet Host Number and the last byte is always zero.

SourceTransType and *DestTransType*

DECnet Phase IV protocol type, which has the following values:

14 =	data + discard	11 =	router hello
6 =	data	9 =	level 2 routing
7 =	level 1 routing	13 =	endnode hello

SourceTransAddress and *DestTransAddress*

Always zero for DECnet.

3.5. Novell IPX attributes

SourcePeerType and *DestPeerType* = 11

SourcePeerAddress and *DestPeerAddress*

IPX network numbers of the flow's two hosts, written as four decimal bytes separated by dots and entered in the same way, e.g. 130.216.0.28. Novell network numbers are assigned by the network administrator; at the University of Auckland we use a Novell Server's IP address as its network number.

A full IPX host address is the combination of its network number and Ethernet address. Because the default configuration of NeTraMet limits peer addresses to a maximum of four bytes, it can in general only handle IPX network numbers. For a host on the same network segment as its server the adjacent address provides that host's Ethernet address but this is not the case for IPX packets from another segment, which have the adjacent address of the router through which they arrived.

NeTraMet can also use full (10-byte) IPX addresses. This is implemented by the compile-time option FULL_IPX; you will need to change the *make* files and then recompile NeTraMet and NeMaC to do this.

NeTraMet understands four forms of encapsulation for IPX packets: Raw 802.3 (with FF FF in the two bytes following the frame length), 802.2 (with SSAP = 0E and DSAP = 0E), 802.2 SNAP and Ethernet II (with type = 8137). Raw 802.3 is the default encapsulation for Novell 3.xx.

SourceTransType and *DestTransType*

XNS protocol type. Possible values are given in RFC 1700, e.g:

1 =	routing information	5 =	sequenced packet
4 =	packet exchange	17 =	Netware Core Protocol

SourceTransAddress and *DestTransAddress*

Source and destination IPX port numbers. These include

1105 =	NCP (Netware Core Protocol)
1106 =	SAP (Service Advertising Protocol)
1107 =	RIP (Routing Information Protocol)

Further details of Netware communications protocols are given in "Netware Communications Processes."

3.6. EtherTalk attributes

SourcePeerType and *DestPeerType* = 12

SourcePeerAddress and *DestPeerAddress*

AppleTalk addresses of the flow's two hosts, written as four decimal bytes separated by dots and entered in the same way, e.g. 0.129.251.0. The first two bytes are the host's AppleTalk (16-bit) network number, the third is its node number (dynamically assigned when it starts up) and the fourth is always zero.

SourceTransType and *DestTransType*

AppleTalk DDP protocol type. Some common values are:

1 = RTMP	5 = RTMP
2 = NBP	6 = ZIP
3 = ATP	7 = ADSP
4 = EP	

SourceTransAddress and *DestTransAddress*

Source and destination AppleTalk socket numbers. These are two-byte numbers; the socket numbers are the low-order byte and the high-order byte is zero.

Details of the AppleTalk protocols are given in the book "Inside Macintosh."

3.7. CLNS attributes

SourcePeerType and *DestPeerType* = 3

CLNS is implemented by the compile-time option CLNS, which is set by default. The large size of NSAP addresses reduces the number of flows which will fit into PC memory – if you don't want NeTraMet to handle CLNS you will need to change the *make* files and then recompile NeTraMet and NeMaC.

SourcePeerAddress and *DestPeerAddress*

NSAP addresses of the flow's two hosts, written as (up to) 20 hexadecimal bytes separated by hyphens. They may be entered in this form or as decimal bytes separated by dots.

SourceTransType and *DestTransType*

CLNS packet type. The most common value is 28, which means 'data.'

SourceTransAddress and *DestTransAddress*

Always zero for CLNS.

Details of the CLNS protocol are given in ISO standard 8473.

3.8. 'Other' attributes

SourcePeerType and *DestPeerType* = 6

Packets for protocols other than those described in sections 3.3 to 3.7 are handled by NeTraMet as 'other' packets. Their Peer attributes are described below, and their Transport attributes are all set to zero.

Rule files may be written to count 'other' packets so as to discover what types of traffic are running on a network segment. If you do this, be aware that the PC NeTraMet generates 'dummy' packets when the network is quiet (it uses these to measure the PC processor utilisation). 'Dummy' packets are a special case of 'other' packets, so your rule file should test for 'dummy' packets and discard them before it tests for 'other' packets.

SourcePeerAddrress

Ethernet II type field from the packet, written as two hexadecimal bytes separated by a hyphen.

DestPeerAddress

LSAP field from the packet header (for 802.3 packets), zero for other packets. Written as two hexadecimal bytes separated by a hyphen.

3.9. General attributes

General attributes are those which relate to a traffic flow itself, rather than to its end-point addresses, and are available for all flows. They include:

SourceInterface, DestInterface
: Interfaces corresponding to the flow's source and destination adjacent addresses. The interfaces NeTraMet is to monitor are specified using command-line options; interface 1 is the first one specified, 2 the second, and so on.

SourceClass, DestClass, FlowClass
SourceKind, DestKind, FlowKind
: These six attributes are not extracted from the packet. Instead they are set by the meter during packet processing by executing *PushRuleto* actions in the rules. They allow a rule set to save information which has been built up during packet matching. For example *SourceClass* and *DestClass* could be set to indicate whether the *source* and *dest* are local, national or international host addresses.

FlowIndex
: (1-origin) index of the flow within NeTraMet's table of flows.

FlowRuleSet
: Number of the rule set the meter was using when the flow was observed.

ToOctets, FromOctets
: Number of bytes observed in the 'to' (source to destination) and 'from' direction for this flow.

ToPDUs, FromPDUs
: Number of packets observed in the 'to' (source to destination) and 'from' direction for this flow.

FirstTime
: Time (in 1/100 second ticks from the time the meter started executing) at which this flow was first observed by the meter.

LastTime
: Time (units as above) at which a packet was last observed for this flow.

MatchingStoD
: This is an attribute which belongs to the meter itself rather than to a flow. Its value is 1 when a packet is being matched in Source-to-Destination ('wire') order.

4. Flow Data Files

By default, NeMaC produces files of flow data information with names like *ccu2.flows.007*. This would be the seventh file of flow data collected from NeTraMet running on the host *ccu2*. Before opening a flow data file NeMaC inspects its current working directory and selects the lowest sequence number not already used for this purpose. Alternatively the user may specify the name(s) of the flow data file(s) NeMaC is to write.

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 separated by separators (if these were specified – see the Format Statement section below) or spaces, and terminated by a newline.

Since Version 2.2 the NeTraMet distribution has included two utility programs which process flow data files:

fd_filter	Computes data rates, i.e. the differences between successive samples in a flow data file
fd_extract	Creates a simple 'column list' file from a flow data file for input to other programs, e.g. gnuplot

These programs are fully described in the *fd_utils* manual.

4.1. Information Records

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. Note that any separators specified in the Format statement appear in the data file directly, not as C-language strings.

Version 4.1 of NeTraMet allowed the meter to run multiple rule sets; this means that the 'rule set number' attribute indicates a row in the meter's RuleInfo table, which is chosen at random by NeMaC when it downloads a rule set. To allow Analysis Applications to associate the rule set numbers in a flow data file with the rule sets which produced them, a new information record was added, the 'Ruleset' record. The format of this is

#Ruleset: *nn setname rfname owner*

The fields are

nn	rule set number, as it appears in flow data records
setname	Name of the rule set, from its SET statement. For v3 and v4 this can only be an integer
rfname	Name of the rule file, e.g. rules.x_ip
owner	Owner name for this rule set

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.

If meter statistics were requested, they appear in a #Stats: record following the #Time: one. The statistics are given as a list of variable names and corresponding values. The variable names are:

aps =	average packets/second	frc =	flows recovered
apb =	average packet backlog	gci =	garbage collection interval (seconds)
mps =	maximum packets/second	rpp =	rules matched per packet
mpb =	maximum packet backlog	tpp =	counts per packet
lsp =	number of packets lost	cpt =	compares per count
avi =	average processor idle %	tts =	total count tables allocated
mni =	minimum processor %	tsu =	count tables in use
fiu =	flows in use		

After its data records the end of each data set is indicated by an #EndData record.

4.2. 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 1995
#Format: flowruleset flowindex firsttime  sourcepeertype sourcepeeraddress
destpeeraddress  topdus frompdus  tooctets fromoctets
#Time: 12:31:27 Wed  1 Feb 1995 130.216.14.251 Flows from 1 to 3642
#Stats: aps=478 apb=11 mps=636 mpb=48 lsp=0 avi=97.3 mni = 93.4 fiu=44 frc=0
gci=5 rpp=1.9 tpp=0.0 cpt=1.0 tts=1024 tsu=38
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 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
#EndData
#Time: 12:36:25 Wed  1 Feb 1995 130.216.14.251 Flows from 3641 to 33420
#Stats: aps=349 apb=16 mps=1357 mpb=537 lsp=0 avi=97.3 mni = 93.4 fiu=480
frc=0 gci=5 rpp=2.4 tpp=1.2 cpt=1.2 tts=1024 tsu=328
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
#EndData
#Time: 12:41:25 Wed  1 Feb 1995 130.216.14.251 Flows from 33419 to 63384
#Stats: aps=415 apb=17 mps=1780 mpb=542 lsp=0 avi=97.3 mni = 93.4 fiu=567
frc=0 gci=5 rpp=1.8 tpp=1.0 cpt=1.3 tts=1024 tsu=372
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
      .  .  .  .  .  .  .  .  .
```

4.3. Flow Data Features

Several features of the Flow data 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. This will happen when the meter switches from a 'current' rule set to its 'standby' rule set.
- *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 will require 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, packet counts and byte counts.

5. Writing Rule Files

5.1. Introduction

A *rule file* is a file of ASCII text which contains information needed by an accounting meter and by a collector. This includes a rule set number, a rule table, a format specification and a statistics request. A rule set is simply a table of rules, identified by its rule set number. An accounting meter can have up to ten rule sets in memory, allowing its manager to switch between them simply by setting the value of the *CurrentRuleSet* MIB variable.

NeTraMet has one rule set built in; it is the default rule set, which is set number 1. This allows NeTraMet to be active as soon as it starts up, and it provides a default rule set which it can use while other rule sets are downloaded by its manager. The default rule set can't be changed by the manager.

NeTraMet version 4.2 introduced SRL, the Simple Ruleset Language and its compiler. SRL is a well-structured high-level language for creating rulesets – one writes an SRL program to specify the flows to be metered and the attributes to be collected for them. The SRL compiler produces flow data files which NeMaC can read and download to a NeTraMet meter. The rest of this chapter describes these rule files, but it is much easier to create them using SRL than it is to do so by hand. See the SRL Manual for more information.

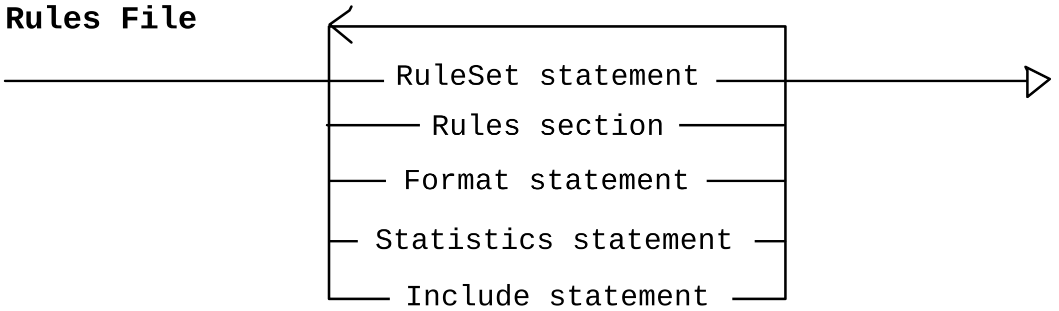
5.2. Rule file Syntax

The syntax for rule files is given below in the form of railway diagrams, and detailed examples are given in the following sections. Note that NeMaC's parser is extremely simple-minded – although it does a good job on valid rule files it has very poor error recovery!

Each *statement* in a rule file starts at the beginning of a line and ends with a semicolon. A cross-hatch character marks the end of a line; all characters following a cross-hatch on a line are ignored by the scanner.

NeMaC's scanner looks for keywords, numbers and addresses. Keywords are shown in the railway diagrams in upper case, but case is ignored by the scanner. Keywords, including attribute names, must be given in full – abbreviations are not allowed.

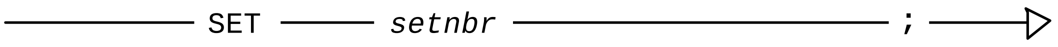
Rules File



A rule file contains one or more of five possible elements, which may appear in any order in the file. The case of characters is not significant within rule files.

5.2.1. RuleSet Statement

RuleSet statement

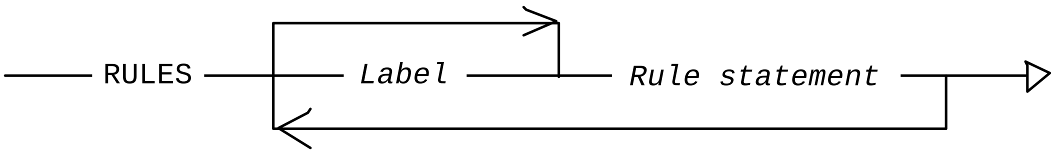


The RuleSet statement supplies a name for the rule set, which can be seen when a meter's rUleset table is displayed.

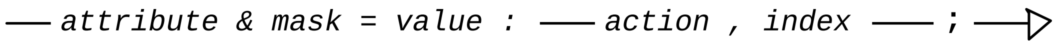
In versions 2 and 3 the RuleSet statement specified which entry in the meter's ruleset table this rule set would occupy.

In version 4.1 the entry is chosen randomly. NeMaC reads it from the meter and puts it into a #Ruleset record of flow data files.

5.2.2. Rules Section



Rule statement



The Rules section specifies the rule table for a rule set, and requests NeMaC to download it to the meter.

It starts with the keyword RULES, followed by a series of Rule statements, one for each rule. Each rule has five components, which must appear in the correct order. These are:

Attribute

The name of the attribute to be tested by this rule. Any of the address attributes may be used, but not the mask or general attributes. The *Null* attribute may also be used, in which case the rule will always succeed.
In addition a meter 'variable' may be tested. There are five of these, named v1, v2, .. v5, each of which can hold the name of an address attribute. A variable's value is set by an 'assign' action (see below).
When a variable appears as the attribute of a rule, its value specifies the PDU attribute to be tested. e.g. if v1 had been assigned *SourcePeerAddress* as its value, a rule with v1 as its attribute would actually test *SourcePeerAddress*.

Mask

Specifies a mask which is ANDed with the attribute's value from an incoming packet. Must be the same length (number of bytes) as the attribute value.
Note that if a rule's mask is zero it will always succeed. This can be useful for rules with 'assign' actions, since it allows them to have meter variables for their attributes.

Value

Specifies the value to be compared with the masked value from an incoming packet. If the compare fails the next rule is tested, otherwise the rule's *Action* is performed.

Action

Action to be performed if the rule's value is matched. Possible actions are:

Ignore	Stop rule matching and return a 'succeed' result. This means the incoming packet will not be counted, i.e. it will be ignored.
NoMatch	Stop rule matching and return a 'fail' result. This allows the meter to interchange the source and destination attributes then retry the match. 'Retry' may be used as a synonym for 'NoMatch.'
Count	Count this packet in the count table specified by this rule's *Index.* Attribute and mask values for a flow are taken from the 'pattern stack,' in the same order as they were pushed during rule processing. The user must ensure that every flow in a count table uses the same set of masks, i.e. was created by the same sequence of PushRuleto and/or PushPktto actions. The attribute values will of course be different for each counted flow.
CountPkt	Executes a PushPktto action (see below), then a Count action (see above).
Return	Return from a rule-matching subroutine, using this rule's *Index* as an offset. The return rule index is popped from the 'return' stack, and the index is added to it, specifying a 'target' rule. The target rule's *Action* is executed immediately, as though that rule's test had just succeeded. The overall effect is that rule-matching subroutines may return a result as well as pushing matched attributes onto the pattern stack.
GoSub	Call a rule-matching subroutine. The index of the next rule to be tested is set to this rule's *Index*, and the meter pushes this rule's *Index* onto a 'return' stack.
GoSubAct	Same as GoSub, except that the target rule's *Action* is executed immediately. It's test is assumed to have succeeded.
Assign	The attribute value specified by this rule's *Value* is assigned to the variable specified by this rule's *Attribute*. Note that since Assign and AssignAct use the rule *Value* in an unusual way they should never perform a test; instead they should always be executed as the result of a Return, GoSubAct. AssignAct, GotoAct, PushRuletoAct or PuskPkttoAct action.
AssignAct	Same as Assign, except that the target rule's *Action* is executed immediately. It's test is assumed to have succeeded.
GoTo	Set the index of the next rule to be tested to this rule's *index.*
GoToAct	Same as Goto, except that execution of the target rule starts with its action, not its test.

PushRuleto	Save this rule's attribute name, mask and value on the 'pattern stack' and set the index of the next rule to be tested to this rule's *Index.* The pattern stack records the rules which were correctly matched; this information is used by a count action to construct flows
PushRuletoAct	Same as PushRuleto, except that execution of the target rule starts with its action, not its test. For compatibility with earlier versions NeMaC treats Pushto and PushtoAct as synonyms for PushRuleto and PushRuletoAct.
PushPktto	Save this rule's attribute name and mask, and the masked value from the packet on the 'pattern stack.' Set the index of the next rule to be tested to this rule's *Index. Value* will always be zero, hence PushPktto and PushPkttoAct rules should never perform a test; instead they should always be executed as the result of a Return, GoSubAct. AssignAct, GotoAct, PushRuletoAct or PuskPkttoAct action.
PushPkttoAct	Same as PushPktto, except that execution of the target rule starts with its action, not its test.

Index

This is a parameter for *Action;* see above for its various uses.

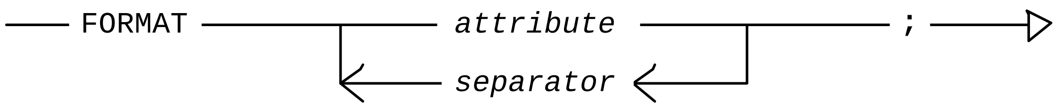
Rule numbers are 1-origin indexes to their corresponding tables. These numbers may be used directly, but it can be difficult to do this accurately. The simplest way to do this is to include the rule number in a comment attached to each statement.

Rules may also be labelled and referred to by their labels, which is very much easier! A label must appear at the beginning of a rule file line, and may contain characters, digits and underscores. NeMaC checks labels for consistency, flagging missing or duplicated labels.

Where a label is needed for 'the next rule' the reserved word *next* may be used, removing the need to create a label.

5.2.3. Format Statement

Format statement



The Format statement specifies the format of rule data lines in a NeMaC Flow Data file.

It starts with the FORMAT keyword, which is followed by a list of flow attributes, in the order they are to appear in the Flow Data file.

5.2.4. Statistics Statement

Statistics statement

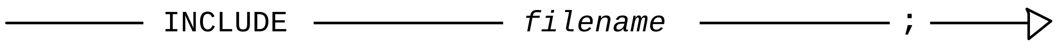
STATISTICS
;

The Statistics statement tells NeMaC to collect meter performance statistics each time it collects flow data, and to write it to the flow data file. NeMaC sets NeTraMet's statistics variables to zero after reading their values.

Caution: since NeTraMet zeroes the statistics each time they are read, only one of the rule sets running on a meter should have a Statistics statement!

5.2.5. Include Statement

Include statement



An include statement allows a rule file to use other rule files. When the NeMaC parser encounters one it saves its position in the current rule file and switches to the file specified. When it reaches the end of the included file it switches back and continues with the earlier file. Include files may be nested up to five deep.

It is useful to have rule-matching subroutines, especially large or complicated ones, in separate files, so that they can be easily used in many other rule files.

5.3. Rule Files

This section gives a few examples of rule files, with comments on what they do and why they were written as they are.

I have found it simplest, when developing a new rule file, to proceed as follows:

- Use labels for any rule which is referred to by another rule. This is much easier than using explicit rule numbers. Remember that the reserved word *next* means 'the next rule,' and use it to avoid labelling consecutive rules.
- Check the rule file by using NeMaC's syntax check (-s -v and -l) options, piping NeMaC's output to a file. Compare the syntax check output with the rule file using two windows in your favourite file editor.

5.3.1. rules.default

```
#  1445, Mon 9 Jan 95
#
#  Default rule file for NeTraMet (built in to the meter)
#
#  Nevil Brownlee,  Computer Centre,  The University of Auckland
#
SET 1
#
RULES
  SourcePeerType & 255 = dummy:  Ignore, 0;  # Ignore meter's dummy pkts
  Null & 0 = 0:  GotoAct, Next;
  SourcePeerType & 255 = 0:   CountPkt, 0;
#
# end of file
```

This is the default rule set, which is built in to the meter and can't be changed. It provides rule set 1 which produces a set of flows, one for each of the peer types which NeTraMet understands.

The first line throws away dummy packets. Dummy packets are generated and processed by the meter when it has no real work to do; the proportion of the time the meter spends processing dummy packets is its measured % idle time.

The second rule is needed so that the third rule's test is never executed. It tests the *Null* attribute, which always succeeds, so the GotoAct action is always executed. If the second rule was not there, *SourcePeerType* would be compared with 0. That test would fail, and since there are no more rules, no packets would ever match.

5.3.2. rules.sample

```
#  1412, Thu 9 Feb 95
#
#  Rule specification file to tally IP net <-> IP net,
#     tally DECnet and Novell and aggregate EtherTalk
#
#  Nevil Brownlee,  Computer Centre,  University of Auckland
#
SET 2
#
RULES
  SourcePeerType & 255 = IP:         Pushto, IP_pkt;
  SourcePeerType & 255 = Novell:     Pushto, Novell_pkt;
  SourcePeerType & 255 = EtherTalk:  Pushto, Apple_pkt;
  SourcePeerType & 255 = DECnet:     Pushto, DEC_pkt;
  Null & 0 = 0:                      Ignore, 0;
```

The first part of this rule table determines the peer protocol type. IP, Novell, EtherTalk and DECnet packets cause the meter to push their *SourcePeertype*, then jump to their protocol's labelled group of rules. The last rule above Other protocols are discarded by the 'Ignore' rule.

```
IP_pkt:
  SourcePeerAddress & 192.0.0.0 = 192.0.0.0:     Goto, low_C;
  SourcePeerAddress & 192.0.0.0 = 128.0.0.0:     Goto, low_B;
  SourcePeerAddress & 192.0.0.0 = 64.0.0.0:      Goto, low_A;
  SourcePeerAddress & 192.0.0.0 = 0.0.0.0:       Goto, low_A;
  Null & 0 = 0:                                  GotoAct, other;
low_A:
  DestPeerAddress & 192.0.0.0 = 192.0.0.0:  GotoAct, A_C;
  DestPeerAddress & 192.0.0.0 = 128.0.0.0:  GotoAct, A_B;
  DestPeerAddress & 192.0.0.0 = 64.0.0.0:   GotoAct, A_A;
  DestPeerAddress & 192.0.0.0 = 0.0.0.0:    GotoAct, A_A;
  Null & 0 = 0:                             GotoAct, other;
low_B:
  DestPeerAddress & 192.0.0.0 = 192.0.0.0:  GotoAct, B_C;
  DestPeerAddress & 192.0.0.0 = 128.0.0.0:  GotoAct, B_B;
  DestPeerAddress & 192.0.0.0 = 64.0.0.0:   GotoAct, B_A;
  DestPeerAddress & 192.0.0.0 = 0.0.0.0:    GotoAct, B_A;
  Null & 0 = 0:                             GotoAct, other;
low_C:
  DestPeerAddress & 192.0.0.0 = 192.0.0.0:  GotoAct, C_C;
  DestPeerAddress & 192.0.0.0 = 128.0.0.0:  GotoAct, C_B;
  DestPeerAddress & 192.0.0.0 = 64.0.0.0:   GotoAct, C_A;
  DestPeerAddress & 192.0.0.0 = 0.0.0.0:    GotoAct, C_A;
  Null & 0 = 0:                             GotoAct, other;
```

This part of the rule table handles IP packets. The first two bits of the packet's source peer address are examined to decide whether it is class A, B or C. For each of these cases the first two bits of the destination peer address is tested to determine its address class. Each of the nine possible pairs of address classes causes a jump to a pair of rules (below) which pushes the packet's source and destination peer addresses. Any other addresses, such as multicast addresses, cause a jump to the rules labelled 'other,' where its full (32-bit) source and destination addresses are pushed. Once the source and destination addresses have been pushed, the rule labelled 'count_pkt' counts the packet.

```
other:
  SourcePeerAddress   & 255.255.255.255 = 0:   PushPkttoAct, Next;
  DestPeerAddress     & 255.255.255.255 = 0:   PushPkttoAct, count_pkt;
A_C:
  SourcePeerAddress     & 255.0.0.0     = 0:   PushPkttoAct, Next;
  DestPeerAddress       & 255.255.255.0 = 0:   PushPkttoAct, count_pkt;
A_B:
  SourcePeerAddress     & 255.0.0.0     = 0:   PushPkttoAct, Next;
  DestPeerAddress       & 255.255.0.0   = 0:   PushPkttoAct, count_pkt;
A_A:
  SourcePeerAddress     & 255.0.0.0     = 0:   PushPkttoAct, Next;
  DestPeerAddress       & 255.0.0.0     = 0:   PushPkttoAct, count_pkt;
```

```
B_C:
  SourcePeerAddress     & 255.255.0.0   = 0:   PushPkttoAct, Next;
  DestPeerAddress       & 255.255.255.0 = 0:   PushPkttoAct, count_pkt;
B_B:
  SourcePeerAddress     & 255.255.0.0   = 0:   PushPkttoAct, Next;
  DestPeerAddress       & 255.255.0.0   = 0:   PushPkttoAct, count_pkt;
B_A:
  SourcePeerAddress     & 255.255.0.0   = 0:   PushPkttoAct, Next;
  DestPeerAddress       & 255.0.0.0     = 0:   PushPkttoAct, count_pkt;
C_C:
  SourcePeerAddress     & 255.255.255.0 = 0:   PushPkttoAct, Next;
  DestPeerAddress       & 255.255.255.0 = 0:   PushPkttoAct, count_pkt;
C_B:
  SourcePeerAddress     & 255.255.255.0 = 0:   PushPkttoAct, Next;
  DestPeerAddress       & 255.255.0.0   = 0:   PushPkttoAct, count_pkt;
C_A:
  SourcePeerAddress     & 255.255.255.0 = 0:   PushPkttoAct, Next;
  DestPeerAddress       & 255.0.0.0     = 0:   PushPkttoAct, count_pkt;
#
count_pkt:
  Null & 0 = 0:  Count, 0;  # Source and Dest Peer Address pushed above
```

Note that the above rules make no attempt to impose any order on source and destination addresses. When a packet appears which is the first of a new traffic flow, its source will be the source for the flow.

```
Novell_pkt:
  SourcePeerAddress  & 255.255.255.255 = 0: PushPktToAct, Next;
  DestPeerAddress    & 255.255.255.255 = 0: CountPkt, 0;
#
Apple_pkt:
  Null & 0 = 0:  Count, 0;  #  No detail for Ethertalk
#
DEC_pkt:
  SourcePeerAddress  & 255.255.255.0 = 0: PushPktToAct, Next;
  DestPeerAddress    & 255.255.255.0 = 0: CountPkt, 0;
```

These actions aggregate EtherTalk and tally Novell and DECnet packets. The tallies push the entire peer address in each case. This instructs the meter to create flows for every possible pair of peer addresses.

```
STATISTICS
#
FORMAT FlowRuleSet FlowIndex FirstTime "  "
   SourcePeerType SourcePeerAddress DestPeerAddress "  "
   ToPDUs FromPDUs "  " ToOctets FromOctets;
#
# end of file
```

The Format statement specifies the attributes to be collected from the meter. It uses double spaces to separate the attributes into four groups.

5.3.3. rules.gateway

```
#  1700, Wed 8 Feb 95
#
#  Rule specification file to tally traffic to/from ccr1
#
#  Nevil Brownlee,  Computer Centre,  University of Auckland
#
SET 3
#
RULES
  DestAdjacentAddress & FF-FF-FF-FF-FF-FF = AA-00-04-00-F4-ED:
     Goto, gateway;
  Null & 0 = 0: Retry, 0;
gateway:
  SourcePeerType & 255 = IP: Pushto, ip_pkt;
  Null & 0 = 0: Ignore, 0;
#
ip_pkt:
  SourcePeerAddress & 255.255.0.0 = 130.216.0.0: Goto, low_C;  # Auckland
```

```
    SourcePeerAddress & 192.0.0.0 = 192.0.0.0:     Goto, low_C;
  low_B:
    DestPeerAddress & 192.0.0.0 = 128.0.0.0:  GotoAct, B_B;
    DestPeerAddress & 192.0.0.0 = 192.0.0.0:  GotoAct, B_C;
    Null & 0 = 0:                             GotoAct, B_A;
```

This rule set was intended for metering traffic through a gateway Ethernet, which had a router with Ethernet Ethernet address AA-00-04-00-F4-ED providing our connection to the Internet. The first rule above tests each packet's *DestAdjacentAddress* to see whether its destination is on the far side of the router. If it is, the rule labelled *gateway* will be tested next, otherwise the 'retry' rule is executed, allowing the meter to interchange source and destinations and try again. If the match fails on the second try the packet is ignored.

The rest of this rule file is very similar to rules.sample above.

5.3.4. rules.broadcast

```
  #  1445, Wed 8 Feb 95
  #
  #  Rule specification file to tally broadcast packets
  #
  #  Nevil Brownlee,  Computer Centre,  University of Auckland
  #
  SET 4
  #
  RULES
    DestAdjacentAddress & FF-FF-FF-FF-FF-FF = FF-FF-FF-FF-FF-FF:
       GotoAct, broadcast;
    Null & 0 = 0 :  Retry, 0;  #  Try other direction
```

This rule set looks for Ethernet broadcast packets, by testing their *DestAdjacentAddress*. Broadcasts are tallied using the group of rules labelled *broadcast* (see below).

The second rule is very important, and is needed because of the way count tables are handled. Consider an incoming packet. If the first matches it is a broadcast packet and is tallied. If this (broadcast) flow is already in the count table it is counted, and the job is done. If it's not, Count doesn't put it in immediately (because it may already be there with its source and destination swapped), and the match fails. The packet is matched again; but rule 1 and rule 2 both fail. In this situation the meter remembers that it was a count which caused the first failure so the match is tried a third time, this time forcing the count to add the flow.

```
  broadcast:
    SourceAdjacentAddress & FF-FF-FF-FF-FF-FF = 0: PushPkttoAct, Next;
    DestAdjacentAddress   & FF-FF-FF-FF-FF-FF = 0: PushPkttoAct, Next;
    SourcePeerType        & 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;
    SourceTransAddress    & 255.255 = 0:           PushPkttoAct, Next;
    DestTransAddress      & 255.255 = 0:           CountPkt, 0;
```

This group of rules pushes all the *adjacent, peer* and *transport* addresses from the packet. Notice that since we don't want any of the tests to be performed (they would fail) we use the 'toAct' form of 'PushPkt.' The last rule pushed the *DestTransAddress* and counts the packet. This example demonstrates how to collect a large amount of detail without explicitly testing each required attribute.

IP protocols BOOTP (ports 67 and 68) and RIP (port 520) are common sources of IP broadcast packets. Remember that ARP is not an IP protocol, so NeTraMet will see ARP packets as 'other' packets.

5.3.5. rules.ipport

```
#  1240, Thu 9 Feb 95
#
#  Rule specification file to tally IP packets by port nbr
#
#  Nevil Brownlee,  Computer Centre,  University of Auckland
#
SET 5
#
RULES
  SourcePeerType & 255 = IP:      Pushto, ip_pkt;
  SourcePeerType & 255 = dummy:   Ignore, 0;  # Ignore meter's dummy pkts
  Null & 0 = 0:                   GotoAct, Next;
  SourcePeerType & 255 = 0:       CountPkt, 0;    # Count packet types
```

IP packets cause the meter to jump to the rule labelled *ip_pkt.* Other packet types are counted by peer type; this is the same as the meter's default rule set (rules.default above).

```
ip_pkt:
  SourceTransType & 255 = tcp:    Pushto, tcp_udp;
  SourceTransType & 255 = udp:    Pushto, tcp_udp;
  SourceTransType & 255 = icmp:   Pushto, c_trans_only;
  SourceTransType & 255 = ospf:   Pushto, c_trans_only;
  Null & 0 = 0:  GotoAct, t_bad;  #  Unknown transport type
#
tcp_udp: s_news:
  SourceTransAddress & 255.255 = nntp:    PushtoAct, c_trans_source;
  DestTransAddress & 255.255 = nntp:      GotoAct, s_news;
s_smtp:
  SourceTransAddress & 255.255 = smtp:    PushtoAct, c_trans_source;
  DestTransAddress & 255.255 = smtp:      GotoAct, s_smtp;
```

The transport type is tested first. TCP and UDP packets jump to the *tcp_udp* label. ICMP and OSPF routing packets jump to *c_trans_only,* where they are tallied by transport type. Unknown packet types jump to *t_bad* where all their attributes are tallied.

```
s_domain:
  SourceTransAddress & 255.255 = domain:  PushtoAct, c_trans_source;
  DestTransAddress & 255.255 = domain:    GotoAct, s_domain;
s_telnet:
  SourceTransAddress & 255.255 = telnet:  PushtoAct, c_trans_source;
  DestTransAddress & 255.255 = telnet:    GotoAct, s_telnet;
s_ftp_ctrl:
  SourceTransAddress & 255.255 = ftp:     PushtoAct, c_trans_source;
  DestTransAddress & 255.255 = ftp:       GotoAct, s_ftp_ctrl;
s_ftp_data:
  SourceTransAddress & 255.255 = ftpdata: PushtoAct, c_trans_source;
  DestTransAddress & 255.255 = ftpdata:   GotoAct, s_ftp_data;
#
  Null & 0 = 0: GotoAct, t_bad;  #  'Unusual' port
#
t_bad:           #  End of packet testing
  SourceTransAddress & 255.255 = 0: PushPkttoAct, Next;
  DestTransAddress   & 255.255 = 0: PushPkttoAct, Next;
  SourceTransType & 255 = 0:    CountPkt, 0;
c_trans_source:  #  SourceTransAddress already pushed
  SourceTransType & 255 = 0:    CountPkt, 0;
c_trans_only:
  SourceTransType & 255 = 0:    CountPkt, 0;
#
FORMAT FlowRuleSet FlowIndex FirstTime "  "
   SourcePeerType
   SourceTransType SourceTransAddress DestTransAddress "  "
   ToPDUs ToOctets  FromPDUs FromOctets;
#
STATISTICS
#
# end of file
```

For each of the specified port types the *DestTransAddress* is tested. If the test succeeds, the *DestTransAddress* (IP port number) is pushed, and the packet is counted at *c_trans_source.*

If that fails, the *SourceTransAddress* is tested. Success there causes the preceding rule's action to be executed. The effect of this is to push the protocol's port number in *SourceTransAddress,* regardless of which direction the packet was travelling.

The overall effect of this rule set is to classify IP traffic into a small group of common traffic types by testing for their well-known port numbers.

5.3.6. rules.manage

```
#  1550, Thu 9 Feb 95
#
#  Rule specification file to tally traffic for Auckland, using four
#     groups of sites: UA-depts, Local, NZ and World
#
#  Nevil Brownlee,  Computer Centre,  University of Auckland
#
SET 6
#
RULES
  SourcePeerType & 255 = IP:   Pushto, IP_pkt;
  Null & 0 = 0 :    Ignore,  0;  #  Ignore other packet types
```

This rule set meters our gateway Ethernet, providing summary information about four groups of IP networks: Auckland University departments, locally-connected networks, New Zealand networks, and the rest of the world.

```
dest_local:
  v1 & 0 = SourcePeerAddress:  AssignAct, Next;
  Null & 0 = 0: Gosub, Auckland_nets;
  Null & 0 = 0:    Goto, c_pkt;  # 1 Dept -> dept-local
  Null & 0 = 0: GotoAct, t_bad;  # 2 UA, not in list of OK subnets -> UA-local
  Null & 0 = 0:      Ignore, 0;  # 3 Local -> Ignore local-local
  Null & 0 = 0:       Retry, 0;  # 4 Not UA or local -> Want local as source
#
dest_UA:
  SourcePeerAddress & 255.255.0.0 = 130.216.0.0: Ignore, 0; # Ignore UA-UA
  Null & 0 = 0 :   Retry, 0;  #  Want Auckland as source
#
IP_pkt:
  DestPeerAddress & 255.255.0.0 = 130.216.0.0: Pushto, dest_UA; # Auckland
  Null & 0 = 0:  GotoAct, Next;
  v1 & 0 = DestPeerAddress:  AssignAct, Next;
  Null & 0 = 0: Gosub, Auckland_nets;
  Null & 0 = 0:             Ignore, 0;  # 1 dest UA department
  Null & 0 = 0:             Ignore, 0;  # 2 dest UA
  Null & 0 = 0:   GotoAct, dest_local;  # 3 dest Local
#
  v1 & 0 = SourcePeerAddress:  AssignAct, Next;  # 4 dest NZ or world
  Null & 0 = 0:  Gosub, Auckland_nets;
  Null & 0 = 0:     GotoAct, src_dept;  # 1 source Dept
  Null & 0 = 0:        GotoAct, t_bad;  # 2 source UA, not an OK subnet
  Null & 0 = 0:    GotoAct, src_local;  # 3 source Local
  Null & 0 = 0:        GotoAct, t_bad;  # 4 Not local, unexpected transit
```

Once a packet has been identified as carrying IP, we reach label IP_pkt. If its destination and source are both Auckland the packet is ignored. If its destination is Auckland but its source is not the match fails; this forces Auckland to be the source of a flow.

The subroutine Auckland_nets is called to test the packet's destination. This returns 1,2,3 or 4 depending on which group the destination belongs to, and pushes the *DestPeerAddress* on the pattern stack.

If the destination is a locally-connected network control passes to Dest_local, where Auckland_nets is called again to determine the packet's source. Local-to-local packets are ignored, Auckland-to-local packets are counted (by going to c_pkt) and packets with a local destination fail, forcing local nets to be the source for flows to nets other than Auckland.

```
src_dept:
  v1 & 0 = DestPeerAddress:   AssignAct, Next;
  Null & 0 = 0:  Gosub, Tuia_proximal;
  DestPeerAddress & 255.255.0 = 130.216.0    : Ignore, 0;     #  1 Auckland
  DestPeerAddress & 255.255.0 = 132.181.0    : Pushto, c_pkt; #  2 Canterbury
  DestPeerAddress & 255.255.0 = 131.203.0    : Pushto, c_pkt; #  3 Gracefield
  DestPeerAddress & 255.255.0 = 141.158.0    : Pushto, c_pkt; #  4 Invermay
  DestPeerAddress & 255.255.0 = 161.65.0     : Pushto, c_pkt; #  5 Lincoln CRI
  DestPeerAddress & 255.255.0 = 130.123.0    : Pushto, c_pkt; #  6 Massey
  DestPeerAddress & 255.255.255 = 192.88.85  : Pushto, c_pkt; #  7 MoRST
  DestPeerAddress & 255.255.0 = 161.29.0     : Pushto, c_pkt; #  8 Mt Albert
  DestPeerAddress & 255.255.255 = 192.122.171: Pushto, c_pkt; #  9 Nat Lib
  DestPeerAddress & 255.255.255 = 192.84.253 : Pushto, c_pkt; # 10 Netway
  DestPeerAddress & 255.255.0 = 139.80.0     : Pushto, c_pkt; # 11 Otago
  DestPeerAddress & 255.255.0 = 160.4.0      : Pushto, c_pkt; # 12 Ruakura
  DestPeerAddress & 255.255.255 = 202.12.76  : Pushto, c_pkt; # 13 Taranaki
  DestPeerAddress & 255.255.0 = 130.195.0    : Pushto, c_pkt; # 14 VUW
  DestPeerAddress & 255.255.0 = 130.217.0    : Pushto, c_pkt; # 15 Waikato
  DestPeerAddress & 255.255.255 = 192.111.102: Pushto, c_pkt; # 16 Wallaceville
  DestPeerAddress & 255.255.0 = 140.200.0    : Pushto, c_pkt; # 17 Tuia
  DestPeerAddress & 255.0.0   = 253.0.0      : Pushto, c_pkt; # 18 Unconnected
  DestPeerAddress & 255.0.0   = 254.0.0      : Pushto, c_pkt; # 19 World
  #
src_local:
  v1 & 0 = DestPeerAddress:  AssignAct, Next;
  Null & 0 = 0:  Gosub, Tuia_proximal;
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; #  1 Auckland
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; #  2 Canterbury
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; #  3 Gracefield
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; #  4 Invermay
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; #  5 Lincoln CRI
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; #  6 Massey
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; #  7 MoRST
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; #  8 Mt Albert
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; #  9 Nat Lib
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; # 10 Netway
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; # 11 Otago
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; # 12 Ruakura
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; # 13 Taranaki
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; # 14 VUW
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; # 15 Waikato
  DestPeerAddress & 255 = 252.0.0 : Pushto, c_pkt; # 16 Wallaceville
  DestPeerAddress & 255 = 253.0.0 : Pushto, c_pkt; # 18 Unconnected
  DestPeerAddress & 255 = 254.0.0 : Pushto, c_pkt; # 19 World
  DestPeerAddress & 255.255 = 140.200: Pushto, c_pkt; # 17 Tuia
```

If the source is an Auckland department or a local network, subroutine Tuia_proximal is called to determine whether the destination is a New Zealand net or not. Tuia_proximal returns values from 1 to 19; for local sources the destination is reduced to either New Zealand (network 252.0.0.0) or World (network 254.0.0.0), but for Auckland departments the destination network (which is the gateway site to which it connects) is pushed onto the pattern stack..

```
c_pkt:
  SourceTransType & 255 = tcp:    Pushto, tcp_udp;
  SourceTransType & 255 = udp:    Pushto, tcp_udp;
  SourceTransType & 255 = icmp:   Pushto, c_trans_only;
  SourceTransType & 255 = ospf:   Pushto, c_trans_only;
  Null & 0 = 0:  GotoAct, t_bad;  #  Unknown transport type
#
tcp_udp: s_news:
  SourceTransAddress & 255.255 = nntp:    PushtoAct, c_trans_source;
  DestTransAddress & 255.255 = nntp:      GotoAct, s_news;
s_smtp:
  SourceTransAddress & 255.255 = smtp:    PushtoAct, c_trans_source;
  DestTransAddress & 255.255 = smtp:      GotoAct, s_smtp;
s_domain:
  SourceTransAddress & 255.255 = domain:  PushtoAct, c_trans_source;
  DestTransAddress & 255.255 = domain:    GotoAct, s_domain;
s_telnet:
  SourceTransAddress & 255.255 = telnet:  PushtoAct, c_trans_source;
```

```
    DestTransAddress & 255.255 = telnet:    GotoAct, s_telnet;
  s_ftp_ctrl:
    SourceTransAddress & 255.255 = ftp:     PushtoAct, c_trans_source;
    DestTransAddress & 255.255 = ftp:       GotoAct, s_ftp_ctrl;
  s_ftp_data:
    SourceTransAddress & 255.255 = ftpdata: PushtoAct, c_trans_source;
    DestTransAddress & 255.255 = ftpdata:   GotoAct, s_ftp_data;
  #
    Null & 0 = 0: GotoAct, t_bad;  #  'Unusual' port
  #
  t_bad:           #  End of packet testing
    SourceTransAddress & 255.255 = 0: PushPkttoAct, Next;
    DestTransAddress   & 255.255 = 0: PushPkttoAct, Next;
    SourceTransType & 255 = 0:    CountPkt, 0;
  c_trans_source:  #  SourceTransAddress already pushed
    SourceTransType & 255 = 0:    CountPkt, 0;
  c_trans_only:
    SourceTransType & 255 = 0:    CountPkt, 0;
```

The rules starting at c_pkt test the transport attributes of IP packets, so as to count separately various types of traffic. ICMP packets are tested for and counted first. News, Mail, Domain and Telnet packets are identified by their port number as either source or destination; transport type is not tested, so these may be either UDP or TCP.

Any packets which don't belong to the categories of interest above are tallied by the t_bad action.

```
  # Auckland local nets
  #
  Auckland_nets:
    v1 & 255.255.255.0 = 130.216.1.0   : PushtoAct, A_dept;  #  Computer Centre
    v1 & 255.255.255.0 = 130.216.3.0   : PushtoAct, A_dept;  #  Auckland DMZ
    v1 & 255.255.255.0 = 130.216.5.0   : PushtoAct, A_dept;  #  Eng Science
    v1 & 255.255.255.0 = 130.216.7.0   : PushtoAct, A_dept;  #  Physics
    v1 & 255.255.255.0 = 130.216.11.0  : PushtoAct, A_dept;  #  Medical School
    v1 & 255.255.255.0 = 130.216.12.0  : PushtoAct, A_dept;  #  Pharmacology
    v1 & 255.255.255.0 = 130.216.14.0  : PushtoAct, A_dept;  #  Commerce
    v1 & 255.255.255.0 = 130.216.15.0  : PushtoAct, A_dept;  #  Mathematics
    v1 & 255.255.255.0 = 130.216.21.0  : PushtoAct, A_dept;  #  Chemistry
    v1 & 255.255.255.0 = 130.216.26.0  : PushtoAct, A_dept;  #  S.B.S.
    v1 & 255.255.255.0 = 130.216.33.0  : PushtoAct, A_dept;  #  Computer Science
    v1 & 255.255.255.0 = 130.216.34.0  : PushtoAct, A_dept;  #  Computer Science
    v1 & 255.255.255.0 = 130.216.73.0  : PushtoAct, A_dept;  #  Law
  #
    v1 & 255.255.255.0 = 192.156.165.0 : PushtoAct, A_local; #  DECUSLINK
    v1 & 255.255.255.0 = 192.251.230.0 : PushtoAct, A_local; #  CLEARFIELD
    v1 & 255.255.255.0 = 202.12.104.0  : PushtoAct, A_local; #  DSE
    v1 & 255.255.255.0 = 202.12.105.0  : PushtoAct, A_local; #  FPNET
    v1 & 255.255.255.0 = 202.14.100.0  : PushtoAct, A_local; #  STATUS
    v1 & 255.255.255.0 = 202.14.102.0  : PushtoAct, A_local; #  KCBBS
    v1 & 255.255.255.0 = 202.14.216.0  : PushtoAct, A_local; #  MANUKAU
    v1 & 255.255.255.0 = 202.14.217.0  : PushtoAct, A_local; #  MALEFICARUM
    v1 & 255.255.255.0 = 202.14.252.0  : PushtoAct, A_local; #  NETBLK-CRAYCOM
    v1 & 255.255.255.0 = 202.14.253.0  : PushtoAct, A_local; #  NETBLK-CRAYCOM
    v1 & 255.255.255.0 = 202.14.254.0  : PushtoAct, A_local; #  NETBLK-CRAYCOM
  #
    v1 & 255.255.0.0 = 156.62.0.0      : PushtoAct, A_local; #  ATINET
    v1 & 255.255.0.0 = 130.216.0.0     : Return,  2; #  University of Auckland
  #
    Null & 0 = 0                       : Return,  4; #  Not dept or local
  #
  A_dept:
    Null & 0 = 0                       : Return,  1; #  UofA department
  #
  A_local:
    Null & 0 = 0                       : Return,  3; #  Auckland local
  #
  # NZ nets (checked by traceroute from Auckland), Tue 19 Oct 93
  #
  # 132.160.0.0   PACCOM
  # 140.200.0.0   KAWAIHIKO
```

```
#
#  Class B nets
#
Tuia_proximal:                                 #   Auckland B
  v1 & 255.255.0.0 = 130.216.0.0    : Return,  1; #  AUCKLAND
  v1 & 255.255.0.0 = 156.62.0.0     : Return,  1; #  ATINET
#                                          Canterbury B
  v1 & 255.255.0.0 = 132.181.0.0    : Return,  2; #  CANTERBURY
  v1 & 255.255.0.0 = 138.75.0.0     : Return,  2; #  LINCOLN-LAN-1

  v1 & 255.255.0.0 = 153.111.0.0    : Return,  2; #  CCCNET2
  v1 & 255.255.0.0 = 165.84.0.0     : Return,  2; #  CHCHPOLY-NET

   .   .   .   .   .   .   .   .   .   .   .   .

#  Class C nets
#                                          Auckland C
  v1 & 255.255.255.0 = 192.156.165.0: Return,  1; #  DECUSLINK
  v1 & 255.255.255.0 = 192.251.230.0: Return,  1; #  CLEARFIELD
  v1 & 255.255.255.0 = 202.12.104.0 : Return,  1; #  DSE
  v1 & 255.255.255.0 = 202.12.105.0 : Return,  1; #  FPNET
  v1 & 255.255.255.0 = 202.14.100.0 : Return,  1; #  STATUS
  v1 & 255.255.255.0 = 202.14.102.0 : Return,  1; #  KCBBS
  v1 & 255.255.255.0 = 202.14.216.0 : Return,  1; #  MANUKAU
  v1 & 255.255.255.0 = 202.14.217.0 : Return,  1; #  MALEFICARUM
  v1 & 255.255.255.0 = 202.14.252.0 : Return,  1; #  NETBLK-CRAYCOM
  v1 & 255.255.255.0 = 202.14.253.0 : Return,  1; #  NETBLK-CRAYCOM
  v1 & 255.255.255.0 = 202.14.254.0 : Return,  1; #  NETBLK-CRAYCOM
#                                          Canterbury C
  v1 & 255.255.255.0 = 192.73.21.0  : Return,  2; #  TUIA-DSIR-1
  v1 & 255.255.255.0 = 192.101.16.0 : Return,  2; #  CHMEDS
  v1 & 255.255.255.0 = 192.122.180.0: Return,  2; #  WAIRCNET

   .   .   .   .   .   .   .   .   .   .   .   .

  v1 & 255.255.255.0 = 202.20.76.0  : Return, 18; #  SANS
  v1 & 255.255.255.0 = 202.20.102.0 : Return, 18; #  AGCTRL
  v1 & 255.255.255.0 = 202.20.103.0 : Return, 18; #  AGLINC
  v1 & 255.255.255.0 = 202.20.104.0 : Return, 18; #  AGGRASS
#
  Null & 0 = 0                      : Return, 19; #  Not an NZ net

#

FORMAT FlowRuleSet FlowIndex FirstTime "  "
   SourcePeerType SourcePeerAddress DestPeerAddress "  "
   SourceTransType SourceTransAddress DestTransAddress "  "
   ToPDUs FromPDUs "  " ToOctets FromOctets;
#
STATISTICS  #  Collect meter statistics
#
# end of file
```

The Format statement specifies the attributes to be collected. These include the transport addresses, allowing for analysis of the traffic by IP service. The meter's performance statistics will also be collected.

5.3.7. rules.lan

```
#  1705, Thu 9 Feb 95
#
#  Rule specification file to tally Local Area Network traffic
#
#  Nevil Brownlee,  Computer Centre,  University of Auckland
#
SET 7
#
RULES
  SourcePeerType & 255 = IP:         PushtoAct, IP_pkt;
  SourcePeerType & 255 = Novell:     PushtoAct, Novell_pkt;
  SourcePeerType & 255 = EtherTalk:  PushtoAct, Apple_pkt;
  SourcePeerType & 255 = DECnet:     PushtoAct, DEC_pkt;
  Null & 0 = 0:                      Ignore, 0;
#
IP_pkt:  # Tally IP traffic by (Class C) subnet
  SourcePeerAddress  & 255.255.255.0 = 0: PushPktToAct, Next;
  DestPeerAddress    & 255.255.255.0 = 0: CountPkt, 0;
#
Novell_pkt:  # Tally Novell traffic by network number and port
  SourcePeerAddress  & 255.255.255.255 = 0: PushPktToAct, Next;
  DestPeerAddress    & 255.255.255.255 = 0: PushPktToAct, Next;
  SourceTransAddress & 255.255         = 0: PushPktToAct, Next;

  DestTransAddress   & 255.255         = 0: PushPktToAct, Next;
  SourceTransType    & 255             = 0: CountPkt, 0;
#
DEC_pkt:
  SourceTransType & 255 = 38: PushtoAct, DEC_hosts;
  SourceTransType & 255 =  6: PushtoAct, DEC_hosts;
  SourceTransType & 255 = 46: PushtoAct, DEC_hosts;
  SourceTransType & 255 = 14: PushtoAct, DEC_hosts;
#
  Null & 0 = 0:  GotoAct, Next  # Tally DECnet non-data packets by type
  SourceTransType & 255 = 0: CountPkt, 0;
#
DEC_hosts:                      # Tally DECnet data by host
  SourcePeerAddress  & 255.255.255 = 0: PushPkttoAct, Next;
  DestPeerAddress    & 255.255.255 = 0: CountPkt, 0;
#
Apple_pkt:
  SourceTransType & 255 = 3: PushtoAct, Apple_hosts;
  Null & 0 = 0:  GotoAct, Next  # Tally EtherTalk by DDP type
  SourceTransType & 255 = 0: CountPkt, 0;
#
Apple_hosts:                    # Tally EtherTalk data by host
  SourcePeerAddress  & 255.255.255 = 0: PushPkttoAct, Next;
  DestPeerAddress    & 255.255.255 = 0: CountPkt, 0;
#
STATISTICS
#
FORMAT FlowRuleSet FlowIndex FirstTime "  "
   SourcePeerType SourcePeerAddress DestPeerAddress "  "
   SourceTransType SourceTransAddress DestTransAddress "  "
   ToPDUs FromPDUs "  " ToOctets FromOctets;
#
# end of file
```

This rule set meters traffic on a busy Local Area Network. It attempts to tally each of four protocols by transport type so as to measure the amount of traffic flowing for each of these. In addition data transport flows are tallied by peer address pairs so as to determine which pairs of hosts generate the greatest proportion of total network traffic.

6. NeMaC Users' Guide

6.1. Overview of NeMaC

NeMaC is a combined manager and collector for the NeTraMet meter. It is a simple Unix program, written in a simple and straightforward way. It is intended to provide control for NeTraMet, so as to make the initial NeTraMet implementation a useful and effective monitoring tool. Later versions will make the coding more elegant, add more features and so on.

If only one meter is to be controlled, all the arguments can be placed on the command line, which is useful when testing new meters and/or rule files. If several meters are to be controlled, default values for the options can be specified on the command line, and the particular parameter values required for each meter can be specified by records in a configuration file.

While NeMaC is running it produces a log file, recording any unusual events observed for any of the meters being controlled. The name of this log file is NeMaC.log.nnn, where nnn is a sequence number starting from 001. When NeMaC starts it scans the current directory for NeMaC log files, then uses the next available sequence number. In this way the log files are preserved through successive runs of NeMaC.

In the same way, when NeMaC starts controlling a meter it opens a 'flows' file. The name of this file is meter-name.flows.nnn. 'meter-name' is the name used to reach the meter via IP; it may be an IP address (e.g. 130.216.234.234) or a host name (provided NeMaC can get its address from a name server). For example the third run of NeMaC controlling meter 130.216.234.234 would produce a flows file called 130.216.234.234.flows.003.

6.2. Command Line Options

NeMaC's command line options are specified as usual, i.e. each option starts with a hyphen, a letter indicating the options, then any parameters required by the option.

If NeMaC is invoked without any options, it will display a summary of the available options.

The options are:

-b mibfile	Gives the name of the MIB file. Default is to read mib.txt from the current directory, or the file specified by the MIBTXT environment variable.
-c nnn	Specifies the required collection interval in seconds. If nnn is zero NeMaC will download rule file(s) to the meter, then exit without collecting any flow data.
-e rulefile	Gives the name of a rule file to be read, downloaded to one or more meters and set up to be those meter's standbyy rule set. Configuration file records may override this for individual meters.
-f cfgfile	Gives the name of NeMaC's configuration file, i.e. the file it will read to determine which meters will be managed and have flow data collected from them. Its default name is NeMaC.cfg.
-g sss	Specifies NeTraMet's garbage collection interval in seconds.
-h pp	Sets NeTraMet's *HighWaterMark* as a percentage of the available flow space, for each manager task, i.e. for every configuration file record.. NeTraMet's default value is 0% - the meter does not test for high water.

-i sss	Specifies NeTraMet's *InactivityTimeout* interval in seconds. NeTraMet's default value is 600 seconds.
-k nnn	Specifies the required keepalive interval in seconds. Every nnn seconds NeMaC will check that the specified meter is running. If it has restarted, NeMaC will download the rule file to it.
-l	Requests NeMaC to list the rule file(s) as they are processed.
-m .ppp	Specifies the UDP port to use for communication with the NeTraMet meter. By default this is port 161 (SNMP).
-o pp	Sets NeTraMet's *FloodMark* as a percentage of the available flow space. NeTraMet's default value is 95%.
-p	For each collection flow data files (as well as log files) will be opened, data appended to them, then they will be closed. If you move or rename a closed flow data file a new one (with the old name) will be created by the next collection. This is a superset of the -P option. This is an alternative to using NeMaC's older 'flag file' method. Still another alternative is to send a SIGUSR1 signal; this will cause NeMaC to start a new flow data file
-r rulefile	Gives the name of a rule file to be read and downloaded to one or more meters. Configuration file records may override this for individual meters.
-s	Tells NeMaC that the rule file is to be read and checked for syntax, but not downloaded to a meter.
-t	This option is for testing – it provides extra diagnostic output from NeMaC.
-u	Specifies that samples should be unsynchronised, i.e. that they should be made every collection interval (-c seconds) after NeMaC starts up.
-v	Asks NeMaC to run in 'verbose' mode. This produces a display of meter status information on the screen at each collection from every meter.
-w nn	Specifies download level. nn = 0 (the default) downloads rules on meter startup and after a meter restart. nn = 1 downloads only after a meter restart, and nn = 2 never downloads.
-x	Don't write anything to the meter. Use this if you use a second copy of NeMaC (or nm_rc) to collect from a single meter. Allowing two collectors to write allows the meter to recover memory for flows after they've been collected by only one of the two collectors.
-E	Specifies the timeout (in seconds) for rEader rows in the meter. If collections stop (e.g. because a manager has failed), the meter will delete the row (i.e. forget about the meter reader) after this time. The default is 0, which means that reader rows will never time out.
-F name	Specifies the name of the flow data file NeMaC is to write.
-L name	Specifies the name of the log file NeMaC is to write.
-P	Open-append-close to NeMaC's log file. This is a subset of the -p option.

-Y name Tells NeMaC to write messages to the Unix syslog. name is used to identify the resulting syslog records. Compile-time variable LOG_LOCAL must be set to enable this option.

Following the options, the name of a meter and its write SNMP community may appear on the command line. In this case, NeMaC will begin managing the specified meter. If NeMaC can find a configuration file, the meter information in that file will override any given on the command line. NeMaC must use a meter's write community, because it sets the meter's *LastCollectTime* variable to tell it that a collection has been started.

For example

```
    NeMaC -c120 -r rules.sample 130.216.234.237 test
```

would cause NeMaC to begin managing meter 130.216.234.237 with write SNMP community 'test'. The rule file 'rules.sample' would be read and downloaded to the meter, and that meter's flow data would be collected every two minutes and written to a file called 130.216.234.237.flows.00x, where 00x was the next available sequence number.

From version 4.1, the NeTraMet meter is able to run more than one rule set at the same time. The meter uses 'Owner Names' to help distinguish its rule sets. You can specify an Owner Name for NeMaC by specifying it on the command line, or any line of a configuration file, after the write community name, e.g.

```
    NeMaC -c300 -r rules.sample my_meter test2 Net-Ops
```

The Owner Name is an alphameric string with a maximum length of 16 characters. It may contain any characters except a blank. In the example above we used `Net-Ops` for NeMaC's Owner Name. If an Owner Name is not specified, 'NeMaC' is used.

The above command would cause NeMaC to begin collecting flow data from meter 'my_meter' with write SNMP community 'test2'. The rule file 'rules.sample' would be read and downloaded to the meter, and that meter's flow data would be collected and written to the flow data file every five minutes.

If two users wish to run NeMaC at the same time, they need to agree to use different Owner Names, otherwise both might use 'NeMaC' by default, which is bound to cause confusion!

When NeMaC is shut down gracefully (by a SIGTERM or SIGINT signal) it shuts down all the tasks it is running on all its meters, without collecting the flow data from those tasks.

When NeMaC execution stops for any other reason, NeMaC cannot stop any of its rule sets executing on any of the meters it is controlling. Instead, they continue to run until NeMaC restarts. At that point, NeMaC reads the meters’ ruleset tables. The 'download level' (-w 1) option can be used to avoid reloading existing rulesets; flows belonging to them will be read. By default (-w 0), NeMaC will delete any rulesets (and the flows they have created) before downloading the rulesets and starting them running.

Again

```
    NeMaC -s -l -r rules.special > syntax.special
```

would cause NeMaC to perform a syntax check on the rule file 'rules.special,' writing a listing of the file during the syntax check. Output from this operation is directed to a file called 'syntax.special' for later inspection.

By default collections are synchronised, i.e. after the first one the next collection time is rounded up to the next multiple of the collection time. e.g. for quarter-hour samples (-c900) they occur at 0, 15, 30 and 45 minutes past each hour. The accuracy of this timing is somewhat limited, and will depend - among other factors - on whether NeMaC has more than one meter to manage and on how many flows each meter has active.

6.3. Configuration File Format

The name of the configuration file is specified by the -f command line option (above). If this option is not used, 'NeMaC.cfg' is used as the default name. If NeMaC can find the configuration file it will read it and start controlling meter(s) as specified in the configuration records; otherwise it assumes that only one meter is to be controlled, and that all options are specified on the command line.

Each record in a configuration file may contain the following options:

-c nnn	Specifies the meter's collection interval in seconds.
-e rulefile	Gives the name of the 'standby' rule file to be used for this meter.
-g sss	Specifies the meter's garbage collection interval in seconds.
-h pp	Sets the meter's *HighWaterMark* for this record as a percentage of the available flow space.
-i sss	Specifies the meter's *InactivityTimeout* interval in seconds.
-k nnn	Specifies the meter's keepalive interval in seconds.
-n nnn	Specifies the meter's *Sampling Rate.* The meter will only process one out of every nnn packets. Default value is 1, i.e. all packets are processed..
-o pp	Sets the meter's *FloodMark* as a percentage of the available flow space.
-r rulefile	Gives the name of the rule file to be used for this meter.

Following the options, the name of a meter and its write SNMP community must appear on the configuration record.

Since NeMaC command-line parameters can displayed by any user via the Unix ps command, you should always specify write community names in a configuration file. Each record in a configuration file specifies meter parameters which override the default values or the ones specified on the NeMaC command line. NeMaC uses the meter name 'default' to indicate that this record contains default values for following records. For example

```
    ./NeMaC -f nm-config
```

tells NeMaC to read the file 'nm-config,' which contains the following records

```
    -c900 -p -rrules.mynet  default
          meter1 write-1
          meter2 write-2
    -c300 meter3 write-3
```

This starts three meters; all run rules.mynet, and append to their flow data files. meter1 and meter2 are to be collected every 15 minutes, and meter3 every 5 minutes.

6.4. Restarting Flow Data Files

Once NeMaC is running it continues to sample its specified meters, producing a corresponding set of flow data files, until NeMaC's execution stops. As part of its operation NeMaC periodically looks to see whether a file called NeMaC.flag exists. If it does, NeMaC closes all its flow data files and opens new ones.

As an example of this, consider a traffic monitoring system which is to produce daily flow data files for a meter called example.meter. If there were initially no flow data files in NeMaC's directory, it would start by creating one called example.meter.log.001, and commence writing samples to it. One could then set up a chron job to be run at midnight every day which renames example.meter.log.001 to something like example.meter.monday, then touches NeMaC.flag. This will create an empty file called NeMaC.flag; when NeMaC sees this it will close the old flow data file (now called example.meter.monday) and open a new flow data file called example.meter.log .001.

An alternative to the above 'flag file' scheme is to use the -P option to tell NeMaC it should close the flag file after writing the data from a collection, then open it again so as to append data to it for later collections. This allows you to ename the flow data file; NeMaC will create a new flow data file for it's next collection.

If you find NeMaC's default method of naming flow data and log files limiting in any way, you can specify the names yourself using the -F and -L options.

7. NeTraMet Users' Guide

7.1. Command Line Options

NeTraMet is started from the command line like any other program. Command line options are specified in a Unix-like way, i.e. each option starts with a hyphen, then a letter indicating the option, and any parameters it requires.

If NeTraMet is invoked without any options, it will display a summary of the available options.

The options are:

-f nnn	Sets the maximum number of flows to nnn. The default for this is 4000; it may be sensible to use a smaller number on the PC if you are using a large rule set.
-i ifn	Tells NeTraMet which interface to monitor. If this is not specified libpcap chooses the default Ethernet network interface. Up to four interfaces may be specified for the Unix and PC meters. For the Unix meter, ifn is the interface name, e.g. le0. For the PC meter it is the software interrupt number in decimal, e.g. 96 = 0x60.
-I ifn	Tells PC NeTraMet to use the specified interface for IP communication, but not to monitor it. If this option is used, up to three other interfaces may be monitored.
-k	Disables the keyboard. If your PC has a BIOS which will start without a keyboard connected, use this option to tell NeTraMet there is no keyboard. If you are running NeTraMet as an unattended background process under Unix you *should* disable the keyboard.
-l	Specifies that the meter should use the length field from IP headers for the number of bytes in an IP packet. Default is to use the MAC (hardware) packet size.

-m ppp	Specifies the UDP port to use for communication with meter readers such as NeMaC. By default this is port 161 (SNMP).
-n nnn	Sets the meter's *Sampling Rate.* The meter will only process one out of every nnn packets. Default value is 1, i.e. all packets are processed.
-p nnn	Sets the size of NeTraMet's buffer for incoming packet headers. The default size is 1024 packet headers.
-r rsc	Specifies a read SNMP community for NeTraMet. Only one read community may be specified.
-s	Disables the screen display. If NeTraMet's screen will never be looked at, it makes sense not to spend processor cycles on maintaining a display.
-u nnn	Allocates space in the meter for a maximum of nnn rules (total for all rulesets in the meter). Default is 2000 rules.
-w wsc	Specifies that NeTraMet's write SNMP community is to be wsc. The default for this is private. Note that the write community must not be the same as the read community.

7.2. PC Screen Display

The display has three main areas; the top left corner is the 'status' area, the bottom left is the 'history' area, and the right-hand half displays a strip chart showing network utilisation.

- The Status Area is updated every second to indicate the time, number of packets (p=), bytes (b=), and utilisation % (u=) for that second. It shows the maximum packet backlog (q=), i.e. the maximum length of the queue of uncounted packets during the second. The meter has buffer space for 1024 packets, so this parameter gives a good indication of the meter's ability to handle the current packet load. The Status Area also shows the active flows % (a=). Since this takes significant resources to compute it is only checked every 30 seconds.
- The History Area displays messages about the meter's operations. These are written on the bottom line of the area, which is then scrolled up one line. The messages are time-stamped, so this area tells you what the meter, manager and collector have been doing recently.
- Every ten seconds a new line of the chart is displayed on the bottom of the strip chart showing the minimum, average and maximum utilisation per second. The minimum is marked with a <, maximum with a > and the average with a *. The scale of the utilisation chart is normally 0 to 30% in 1% steps, but the 'h' keyboard command can be used to halve it, i.e. change it to 0 to 60% in 2% steps. The chart is scrolled up the screen as each line is displayed, so that it always shows the network utilisation for the last 250 seconds.

7.3. Keyboard Commands

If the keyboard is enabled, i.e. the -k option did not appear on its startup command line, pressing a key will perform various functions, as follows:

a	Display 'manager tAsk' information table
b	Display 'bad packets' counts
e	Display 'meter rEader' information table
f	Display flow table statistics. These include active and inactive flows, as well as information about collection times and collector IP addresses.
h	Set/reset half-scale for utilisation strip chart
m	Display meter's memory usage
p	Display list of protocols meter is currently recognising. This will depend on the current rule table
s	Display meter performance statistics. These are explained further in the next section.
t	Display time in 1/100s intervals from meter startup
u	Display 'rUleset' information table
v	Display meter version info
z	Set meter statistics variables to zero
?	Display help (summary of keyboard commands)
ESC	Stop metering, exit to DOS.

7.4. PC Statistics Display

The statistics displayed (console 's' command) are as follows:

```
Meter Statistics ..
  Av pkt/s 275, av pkt backlog 1
  Max pkt/s 818, max pkt backlog 3
  Idle time av 99.1, min 96.5 %
  66 flows active (max 3200)
  0.5 rules/pkt, 0.2 counts/pkt
  1.0 compares/count
```

Meter statistics are computed using counters which are updated every second. These counters can be set to zero by the manager, or by pressing the 'z' key. NeMaC, if instructed by the rule set, can read the statistics variables then set their counters to zero each time it collects the flow data. The statistics are provided so as to evaluate the meter's performance on various hardware configurations, network traffic loads and rule tables. A brief explanation of each is given here.

'Packets per Second' gives the average and maximum packet rates observed since the statistics counters were last set to zero.

'Packet Backlog' refers to the maximum length of the queue of packets received but not yet processed by the meter. The maximum queue length is 1024; packets received when the buffer is full are counted as lost packets, then discarded. The *LostPacket* count can be displayed by pressing the 'b' key.

NeTraMet's highest-priority process attempts to take packets from the input queue, up to a maximum of 400 at a time. This prevents its lower-priority processes such as the keyboard handler from being blocked indefinitely. If there are no packets in the queue a dummy packet is generated and passed to the packet matching routine, where it is counted. The 'Idle Time' measurements are the ratio of dummy packets to total packets (i.e. dummy + real packets) processed by the meter.

'Flows Active' here means flows which currently hold valid flow data which has not been collected. Once flows become inactive their space can be recovered by the garbage collector. If the meter runs out of space for new flows the garbage collector will reclaim the oldest inactive flows first.

'Rules per packet' and 'Counts per Packet' show how many rules were tested and how many count actions were performed for each packet. Counts are implemented using hash tables for the flows; the number of 'Compares per Count' gives an indication of how long the hash chains have become.

7.5. PC Flow Table Information Display

The statistics displayed (console 'f' command) are as follows:

```
Flow Table Info ..
  Collection times:
    130.216.3.1     32 s,     2 s
  InactTime 600, HighWater 65, Flood 95
  CurrentRuleSet 2, EmergencyRuleSet 0
  Flows: 56 active, 94 used, 3200 max
    0 recovered (GC: 10 s, 32 flow)
    Creating   2.000 flow/s
    Recovering 0.000 flow/s
```

'Collection Times' gives information about collectors gathering flow data from this meter. The collector's IP address appears first, followed by the elapsed time since the last collection (2 seconds above) and the collection before that. Inactive flows (older than 32 seconds above) may be recovered by the garbage collector.

The next two lines show the current values of the meter's memory management parameters.

The 'Flows' line shows the maximum number of flows (NeTraMet's -f command-line parameter), the number in use (active and inactive) and the number active.

'Flows Recovered' shows the number of flows reclaimed by the garbage collector since the statistics counters were last set to zero. The garbage collector is controlled by two parameters, which are displayed after the GC: label. The first of these is the interval in seconds between invocations of the garbage collector. Its default value is 5, but it can be changed by the manager. The other parameter are the number of flows tested by the garbage collector each time it is invoked. These cannot be changed by the user.

The last two lines show the rate the flows are created (by the rules in response to incoming packets) and recovered (by the garbage collector).

7.6. Configuring Waterloo TCP for NeTraMet

Waterloo TCP stores its configuration data in a file called WATTCP.CFG. For use with NeTraMet it is simplest to place this file in the same directory as NeTraMet itself

A sample WATTCP.CFG file is included in the NeTraMet distribution. This will need to be edited to specify the IP Address, Subnet Mask, Default Gateway and Domain Name for NeTraMet at the location where you intend to run it. The file is in plain ASCII text, and it is obvious which lines need to be modified.

7.7. Sample AUTOEXEC.BAT file

```
wd8003e  0x60  5 0x300 0xD800
NeTraMet  -r remote  -w Net*Manager
```

The first line above starts the packet driver for a Western Digital Ethernet card, using hardware interrupt (IRQ) 5, I/O address 0x300, shared memory address 0xd800 and packet interrupt 0x60. NeTraMet searches the interrupt vector when it starts up, which allows you to use any valid packet interrupt address. For Versions 2.2,above 2.2,3.5, NeTraMet can handle up to four packet drivers.

The second line starts NeTraMet, specifying that it is to have read SNMP andcommunity 'remote', andthat its write communityis 'Net*Manager'. The screen and keyboard are enabled by default, and the meter will use the default maximum of 4,000 flows.

7.8. Differences between PC and Unix versions of NeTraMet

On the PC a packet driver is used to access each network interface. If no interface is specified NeTraMet uses the first packet driver it finds as the interface to monitor. On Unix the default network interface will be monitored by default. On a PC or Unix system you may use the -i command-line option to specify which interface(s) to monitor.

The PC version of the meter runs on a system with a dedicated screen and keyboard, and hence will respond to keyboard commands, and provides a continuously-updated status display.

The Unix versions are intended to run as background processes on a multi-user system. When starting NeTraMet as a background process, don't forget to use the '-k' option to prevent it trying to read from `stdin.`

In the same way the Unix version doesn't provide a status display. If its screen is enabled it will display history messages as events occur, but that is the full extent of its screen output.

Some of the PC statistics variables have been specifically designed to monitor the hardware performance of the PC, so they are not relevant to Unix. In particular, packet backlog statistics are not implemented for Unix. Other statistics, such as processor utilisation, use Unix system calls to gather the requested information.

Apart from the above comments about screen and keyboard, the two versions are identical. From the performance point of view there is one further feature of the Unix version; it is not limited by the PC's arcane memory models, so that it can handle more flows than the PC (which has a limit of about 4500). This problem was greatly reduced when it became possible to make a 32-bit PC meter – since then both Unix and 32-bit PC meters are limited only by the fact that *FlowIndexes* are 32-bit values!

8. NeTraMet Distribution

8.1. Copyright Statement

NeTraMet is free software, distributed under the terms of the GNU General Public License. A copy of this is provided with the NeTraMet software distribution files.

8.2. Distribution Files

The NeTraMet distribution files include the following:

**.pdf*
    Documentation files for NeTraMet, in Adobe Portable Document Format (PDF)

NeTraMet43.tar.gz
    NeTraMet documentation, including example rule files and the Meter Services MIB
    Source code for CMU SNMP, NeTraMet, nifty, srl, NeMaC, etc.

ntm43-pc.zip
    Executable files for PC NeTraMet meters (32-bit, OCxMON and 16-bit)

ntm43-src.zip
    Source and Make files for PC NeTraMet meters

Solaris.tar.gz
    Solaris binary files for the NeTraMet programs

Irix.tar.gz
    Irix binary files for the NeTraMet programs

Linux.tar.gz
    linux binary files for the NeTraMet programs

9. Installation

On a Unix system:

Create a directory for NeTraMet and place the NeTraMet.tar.gz file there. Uncompress it and unpack it; this will create directories and place NeTraMet's files in them as follows:

```
./README            Introduction           } Read these two files
./INSTALL           Install instructions   }   before continuing !
./documentation/
   NeTraMet/        NeTraMet documentation
   Examples/        rule files
      srl/          SRL programs
   snmp/            CMU SNMP documentation
./mib/              Traffic Meter MIB
./src/
   snmplib/         CMU SNMP library source
   apps/            CMU SNMP applications source
   meter/           NeTraMet source
   manager/         NeMaC source
   srl/             SRL compiler source
```

The Unix meter uses libpcap to observe packet headers. libpcap is available from

```
ftp://ftp.ee.lbl.gov/libpcap-*.tar.gz
```

Install it on your machine, i.e. build the libpcap.a file and either install it as a system file or simply copy it into the NeTraMet's src/meter directory.

There are several compile-time options for the NeTraMet programs, including:

CLNS	Reset by default	Allows NeTraMet to meter CLNS packets
FULL_IPX	Reset by default	Tells NeTraMet to use full (10-byte) IPX peer addresses, instead of just (4-byte) network numbers
V6	Reset by default	Allows NeTraMet to recognise and handle IPv6 packets

These options should be set for all the NeTraMet programs by editing the *configure* file before running it to build the programs, or (if you have some experience with GNU autoconfigure) by editing the *configure.in* file then using autoconfig to create a new *configure* file from it.

Instructions for building the NeTraMet programs are given in the INSTALL file. NeTraMet is built in the meter directory, NeMaC in the manager directory and srl in the srl directory.

Once the programs are built you can begin to use them, as follows ..

Copy NeTraMet and NeMaC to the location where they will be used. NeTraMet opens a UDP port for SNMP; make sure it has sufficient privilege to do this. NeMaC doesn't need special privilege, but it needs access to the mib.txt file (in the /mib directory). Set an environment variable to specify this, e.g.

```
setenv MIBTXT /usr/local/NeTraMet/mib/mib.txt
```

Decide on write community names for each meter you intend to run. Start the meters.

Create rule files for each meter. If there are many of these it will be sensible to create a configuration file with an entry for each of them. Start NeMaC.

On a PC:

Full instructions for setting up the PC versions of NeTraMet are included in the ntm*-pc.zip file, including run-time support files for the 32-bit version. The distribution file includes full source for Joel Apisdorf's OCxMON traffic measurement system; this provides the 32-bit environment for the NeTraMet meters.

If you wish to compile and link NeTraMet yourself, copy ntm*-src.zip to your PC and unzip to create directories and place files into them as follows:

`/wattcp`	Waterloo TCP library source
`/snmplib`	CMU SNMP library source
`/netramet`	NeTraMet source
`/..`	Other OCxMON source files
`/ntm32`	32-bit meter Makefiles
`/ntmoc`	OCxMON meter Makefiles
`/ntm16`	16-bit meter Makefiles

To create one of the versions of NeTraMet cd into the appropriate directory, e.g. ntm32 for the 32-bit meter. Build the snmp and Waterloo TCP libraries, then build the meter as follows:

```
make –f snmp
make –f wattcp
make
```

10. NeTraMet's Future

This document describes the current version of NeTraMet; it will doubtless improve and grow. Please report any bugs or problems you encounter to me directly, n.brownlee@auckland.ac.nz.

There is a NeTraMet Users' mailing list; any comments, suggestions, enquiries, etc will be welcome. To send a message to the list, send it to netramet@auckland.ac.nz. To subscribe to the list send a message to majordomo@auckland.ac.nz with the body

```
subscribe netramet your.address@domain
```

Discussion about the Realtime Traffic Flow Measurement (RTFM) Architecture will continue on the Working Group's mailing list, rtfm@auckland.ac.nz. If you are interested in network traffic measurement please join this list by sending a message to majordomo@auckland.ac.nz with the body

```
subscribe rtfm  your.address@domain
```

The RTFM Working Group maintains a WWW page at

`http://www.auckland.ac.nz/net/Internet/rtfm`

I would appreciate user feedback and reports of your experiences with it, in particular:

- Performance of various configurations of PC and Unix meters on different networks.
- Ports of NeTraMet or NeMaC to other operating systems.
- Developments of programs for processing flow data files, e.g. to produce an input file for a statistics package.
- SRL programs. How hard or easy did you find it to create rule files for your metering requirements using SRL? Were there things you wanted to do but couldn't? Did you discover any particularly elegant programming techniques for metering your network?
- Suggestions for new features in existing NeTraMet programs, or ideas for new programs which could form part of the NeTraMet system.

Please post 'experience' reports and suggestions to the NeTraMet Mailing List .

11. Acknowledgments

Many people have contributed to the development of NeTraMet. I wish to record my thanks particularly to those who participated in the early discussions of the Internet Accounting Working Group, which developed the Internet Accounting Architecture. Thanks to:

Cyndi Mills and Greg Ruth (BBN) *Co-chairs to March 93*
Kathy Robertson (Concord Communications), George Abe (infoNet)
Marshall Rose (Dover Beach Consulting)

NeTraMet is the first implementation of the RTFM Working Group's draft Meter Services MIB. My colleagues here at Auckland have contributed many hours of discussion throughout NeTraMet's development. Special thanks to:

John White, Russell Fulton, Murray Johns
Wilson Yan
Sig HandelMan, Stephen Stibler

Making the OCxMON version of NeTraMet would not have been possible without the support and encouragement of the OCxMON team at MCI, especially

Joel Apisdorf, Rick Wilder

Since NeTraMet's initial release in October 1993 many people have reported bugs, suggested improvements, supplied information about protocols and contributed patches to the source code. Special thanks to:

Nicolai Guba (BT Labs)
Mika Hautanieni (Helsinki University of Technology)
Steven Heitmeyer (Oregon State University)
Kevin Hoadley (JANET)
Jacek Kowalski (Telstra Research)
Scott Marcus (BBN)
Tran Phan Anh (University of Kansas)

12. References

"ISO 8473: Information Processing Systems – Data Communications –
Protocol for providing the Connectionless-mode Network Service," 1992

"RFC 1700: Assigned Numbers," J. Reynolds, J. Postel,
October 1994

"RFC 1272: Internet Accounting: Background," C. Mills, D. Hirsh, G. Ruth,
November 1991

"Internetworking with TCP/IP Vol 1 (2nd Edition), " Comer, D,
Prentice Hall, 1991

"Inside AppleTalk (2nd Edition)," Sidhu, Andrews and Opppenheimer,
Addison Wesley, 1990

"Netware Communications Processes," Netware Application Notes,
September 1990