**INTRODUCTORY DOCUMENTATION**

# NeTraMet & NeMaC

Network Traffic Meter & NeTraMet Manager/Collector

*Version 2.1*

*Nevil Brownlee*

Computer Centre
The University of Auckland
Auckland, New Zealand

January 1994

## 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 1993

# CONTENTS


1. Introduction 1
   - 1.1. Operating Environments 1
   - 1.2. Traffic Flows 1
   - 1.3. The Internet Accounting Meter Services MIB 2
2. Implementation Details 2
   - 2.1. Program Development Environment 2
   - 2.2. Data Structures: Flow, Rule and Action Tables 2
   - 2.3. Meter Memory Management 3
   - 2.4. Optimising the Rule Table 3
   - 2.5. The Meter's Outer Block 4
   - 2.6. Matching the Rules 4
   - 2.7. SNMP: Getting and Setting Variables 4
   - 2.8. Creation and Activity Tables 4
   - 2.9. Column Blobs: Recovering Bulk Flow Data 5
3. Flow Attributes 5
   - 3.1. Introduction 5
   - 3.2. Ethernet attributes 5
   - 3.2. IP attributes 6
   - 3.3. DECnet attributes 6
   - 3.4. Novell IPX attributes 7
   - 3.5. EtherTalk attributes 7
   - 3.6. CLNS attributes 8
   - 3.7. General attributes 8
4. Flow Data Files 9
   - 4.1. Information Records 9
   - 4.2. Sample Flow Data File 9
   - 4.3. Flow Data Features 10
5. Writing Rule Files 11
   - 5.1. Introduction 11
   - 5.2. Rule file Syntax 11
     - 5.2.1. RuleSet Statement 12
     - 5.2.2. Rules Section 12
     - 5.2.3. Actions Section 14
     - 5.2.4. Format Statement 15
     - 5.2.5. Statistics Statement 15
   - 5.3. Rule Files 15
     - 5.3.1. rules.default 15
     - 5.3.2. rules.sample 16
     - 5.3.3. rules.gateway 17
     - 5.3.4. rules.broadcast 18
     - 5.3.5. rules.manage 19
     - 5.3.6. rules.lan 23

6. NeMaC Users' Manual 24
6.1. Overview of NeTraMet 24
6.2. Command Line Options 24
6.3. Configuration File Format 25
7. NeTraMet Users' Manual 25
7.1. Command Line Options 26
7.2. PC Screen Display 26
7.3. Keyboard Commands 26
7.4. PC Statistics Display 27
7.5. Configuring Waterloo TCP for NeTraMet 28
7.6. Sample AUTOEXEC.BAT file 28
7.7. Differences between PC and SunOS versions of NeTraMet 28
8. NeTraMet Distribution 29
8.1. Copyright Statement 29
8.2. Distribution Files 29
9. Installation 29
10. NeTraMet's Future 31
11. Acknowledgments 31
11. References 32

*Nevil Brownlee is the Deputy Director of the University of Auckland Computer Centre, and is responsible for support and development of the University's campus network, which has about 2,500 connected hosts. He is closely involved with Tuia, New Zealand's Research and Education network, co-ordinates Kawaihiko, the New Zealand Universities' group within Tuia, and is active within the IETF, especially on the Internet Accounting 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 Internet Accounting Working Group's Internet Accounting 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;
- collectors, which retrieve information from meters;
- managers, which instruct meters as to which flows they should measure and collectors as to which meters they should collect from, at what intervals.

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

## 1.1. Operating Environments

This release of NeTraMet runs on a SunOS host, using NIT to observe ethernet packet headers, or on a PC using a CRYNWYR packet driver. The PC implementation can be used with a 10 MHz 286 machine, on which it can handle a steady packet load of about 750 packets per second. On a 25 MHz 386SX it will handle about 1250 packets per second. NeTraMet can cope with peak traffic bursts of up to 2250 packets per second for several seconds at a time.

This release also includes NeMaC (NeTraMet Manager/Collector), a combined manager and collector program. It runs on Unix systems, and Make files are included for SunOS and Irix systems.

## 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) |
| detail | (transport layer) |

Since NeTraMet is (so far) only implemented on ethernet, *adjacent addresses* can only 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 *detail 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. Sub-flow details are specified by a set of attribute masks, which make up a *tally action.*

It is also possible to combine several flows together in the meter. Each of the flows to be combined will then have an *aggregate action* which specifies all the attribute values for the aggregate flow.

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 Internet Accounting Meter Services MIB

The Internet Accounting Group produced an Internet Draft describing its proposed Accounting Meter Services MIB, which was assigned the number Experimental.99.

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.

## 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.

### 2.2. Data Structures: Flow, Rule and Action Tables

Host addresses (adjacent, peer and detail) 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 a rule table is allocated as a single block of memory.

Actions are stored in a flow structure, since they have exactly the same set of components as a flow. An action table is an array of pointers to actions.

The first time a flow is observed the meter allocates space for it and assigns it a *FlowIndex.* The addresses of count and tally flows are held in the rule and action tables. Tallies are implemented as hash tables; a hash table's address is held in its action table entry.

## 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 in use and compare it with the *HighWaterMark* variable. If the percentage of flows in use 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 generate an SNMP trap if flow data is lost; this has not yet been implemented. The values of the memory management variables mentioned above can 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 current rule set 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 tally rule, but its flow was not yet present in the tally. 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 tally with its keys in source-to-destination order.

This algorithm means that you can use symmetrical tallies, i.e. tallies 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.

## 2.8. Creation and Activity Tables

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 the Creation and Activity tables, which provide a collector with windows into the flow table, allowing it to inspect only those flows 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. Column Blobs: 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 solves 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 takes this idea a little further. The user specifies which attributes are to be collected using a Format statement in the rule file. NeMaC uses the Format to decide which columns are required, then retrieves column blobs for each attribute starting at the first row of the flow table. The resulting collected flows are written to disk, then the process is repeated starting at the row after the last collected row, and so on.

The maximum column blob size is 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 wish to collect 10 attributes for 1000 flows, this will require only 10 x 20 = 200 SNMP packets To minimise network loading, NeMaC pauses for 90 milliseconds after each SNMP request.

## 3. Flow Attributes

### 3.1. Introduction

A flow's attributes may be conveniently arranged into five groups: adjacent address, peer address, detail 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 Ethernet Attributes section; they are the same for all of the peer address types.

Peer and detail 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 detail address type are *SourcePeerType* and *SourceDetailType*. *SourcePeerType* and *DestPeerType* are synonyms, as are *SourceDetailType* and *DestDetailType*. 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.

The *SourcePeerTypeMask* and *SourceDetailTypeMask* (or *DestPeerTypeMask* and *DestDetailTypeMask*) attributes can be used to specify that the type fields are to be tallied. In this situation a mask value of 255 (all bits on) is the only sensible one.

### 3.2. Ethernet attributes

NeTraMet's initial implementation supports a single ethernet interface. It therefore supports only one adjacent address type, i.e. ethernet, which is the same for both *SourceAdjacentType* and *DestAdjacentType*. Since there is only one adjacent address type, the *AdjacentTypeMask* attributes are not implemented.

The *SourceAdjacentAddress* and *DestAdjacentAddress* attributes are the ethernet 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 actions to tally fields within adjacent addresses. They are written and entered in the same form as adjacent addresses.

### 3.2. IP attributes

*SourcePeerType* and *DestPeerType* = 2

*SourcePeerAddress* and *DestPeerAddress*
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 tallies; in the same form as peer addresses.

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

| | |
|---|---|
| 1 = ICMP | 17 = UDP |
| 6 = TCP | |

These names can be used for them in rule files.

If the *DetailType* is **TCP** or **UDP**, the *SourceDetailAddress* and *DestDetailAddress* contain the flow's source and destination port numbers. Many of their values are given in RFC 1060, Assigned Numbers. Common values are

| | |
|---|---|
| 20 = FTP-DATA | 53 = DOMAIN |
| 21 = FTP | 119 = NNTP |
| 23 = TELNET | 123 = NTP |
| 25 = SMTP | 161 = SNMP |

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

If the *DetailType* is **ICMP**, then *SourceDetailAddress* contains the ICMP type and *DestDetailAddress* 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 |

*SourceDetailMask* and *DestDetailMask*
Like *SourceDetailType*, 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.

### 3.3. DECnet attributes

*SourcePeerType* and *DestPeerType* = 5

*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.

*SourceDetailType* and *DestDetailType*

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 |

*SourceDetailAddress* and *DestDetailAddress*

Always zero for DECnet.

### 3.4. Novell IPX attributes

*SourcePeerType* and *DestPeerType* = 6

*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 three forms of encapsulation for IPX packets: Raw 802.3 (with FF FF in the two bytes following the frame length), Ethernet II packets with type 8137, and 802.2 SNAP packets. Raw 802.3 is the default encapsulation for Novell 3.xx.

*SourceDetailType* and *DestDetailType*

XNS protocol type. Possible values are given in RFC 1060, e.g:

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

*SourceDetailAddress* and *DestDetailAddress*

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 "Netware Communications Processes."

### 3.5. EtherTalk attributes

*SourcePeerType* and *DestPeerType* = 7

*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.

*SourceDetailType* and *DestDetailType*

AppleTalk DDP protocol type. Some common values are:

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

*SourceDetailAddress* and *DestDetailAddress*

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."

## 3.6. 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.

*SourceDetailType* and *DestDetailType*

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

*SourceDetailAddress* and *DestDetailAddress*

Always zero for CLNS.

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

## 3.7. 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, including 'other' flows, i.e. those with *SourcePeerType* and *DestPeerType* = 12. They may be collected from the meter and may be specified in actions, but may not be used in rules. General attributes include:

*SourceInterface, DestInterface*

Interfaces corresponding to the flow's source and destination adjacent addresses. In the current implementation these always have the value 1.

*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 first observed.

*FlowType*

Type of the flow. Values are: 1 = count, 2 = tally, 3 = aggregate.

*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.

# 4. Flow Data Files

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.

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.

## 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.

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 = | tallies per packet |
| lsp = | number of packets lost | cpt = | compares per tally |
| avi = | average processor idle % | tts = | Total tally slots allocated |
| mni = | minimum processor % | tsu = | Tally slots in use |
| fiu = | flows in use | | |

## 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 v2.0:  -c300 -r rules.lan test_meter  4000 flows  starting at
12:31:27 Fri  1 Oct 93
#Format: flowruleset flowindex firsttime  sourcepeertype sourcepeeraddress
destpeeraddress  topdus frompdus  tooctets fromoctets
#Time: 12:31:27 Fri  1 Oct 93 130.216.14.251 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 16 3560  5 4.1.152.0 59.2.189.0  2 2  120 120
      .  .  .  .  .  .  .  .  .
3 42 3560  7 0.128.42.0 0.129.34.0  0 1  0 60
3 43 3560  7 0.128.42.0 0.128.43.0  0 1  0 60
3 44 3560  7 0.128.42.0 0.128.41.0  0 1  0 60
3 45 3560  7 0.128.42.0 0.129.2.0  0 1  0 60
3 46 3560  5 4.1.152.0 59.2.208.0  2 2  120 120
3 47 3560  5 59.3.46.0 4.1.150.0  2 2  120 120
3 48 3560  5 4.1.152.0 59.2.198.0  2 2  120 120
3 49 3560  5 0.0.0.0 59.2.120.0  0 1  0 60
3 50 3664  5 4.1.152.0 59.2.214.0  0 1  0 60
3 51 3664  5 0.0.0.0 4.2.142.0  0 1  0 60
3 52 3664  5 4.1.153.0 59.3.45.0  4 4  240 240
#Time: 12:36:25 Fri  1 Oct 93 130.216.14.251 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
#Time: 12:41:25 Fri  1 Oct 93 130.216.14.251 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. The above shows flows from rule set 1 – the default set – in the first collection, followed by the first flows created by rule set 3 (which has just been downloaded by NeMaC).
- *FlowIndexes* may be reused by the meter once their flows have been recovered by the garbage collector. The combination of *FlowRuleSet, FlowIndex* and *StartTime* are needed to identify a flow uniquely.
- Packet and Byte counters are 32-bit unsigned integers, and are never reset by the meter. Computing the counts occurring within a collection interval 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, pdu counts and packet 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, rule and action tables, a format specification and a statistics request. The term *rule set* means the combination of a rule table and its corresponding action table. 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.

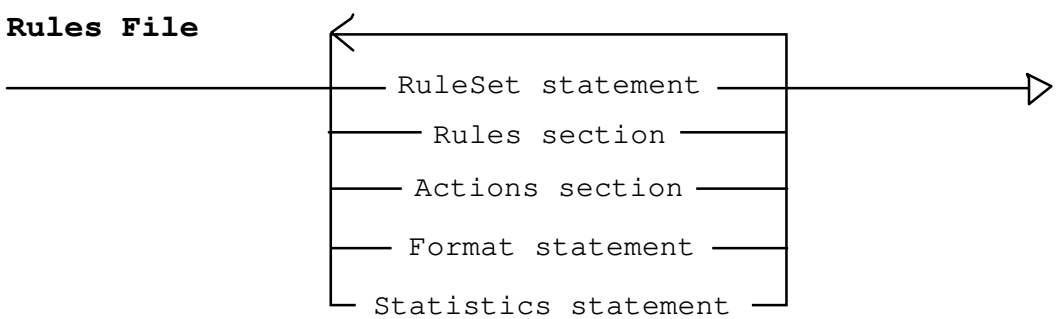
NeTraMet has one rule set built in; this is the default rule set, which is set number 1. This allows NeTraMet to be active as soon as it starts up, and it provides a default rule set which it can use while other rule sets are downloaded by its manager. The default rule set can't be changed by the manager.

## 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.

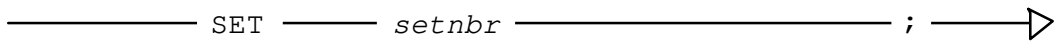
NeMaC's scanner looks for keywords, numbers and addresses. Keywords are shown in the railway diagrams in upper case, but case is ignored by the scanner. Keywords, including attribute names, must be given in full – abbreviations are not allowed.



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.

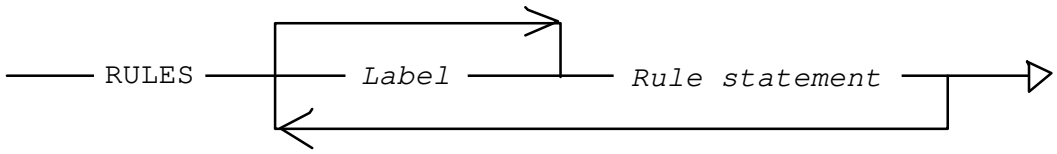
### 5.2.1. RuleSet Statement

**RuleSet statement**

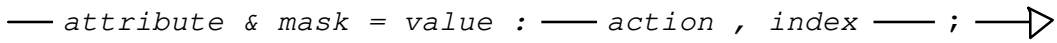



The RuleSet statement tells the meter which rule set it is to use. If a rule file doesn't have a RuleSet statement, the meter will continue to use its current rule set.

### 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:

| | |
|---|---|
| Count | Count this packet in the count action specified by this rule's *Index.* Attribute and mask values for a *count* flow are taken from the matched rules. The user must ensure that every flow in a count action uses the same set of masks – the attribute values will of course be different for each counted flow. Other attributes may be specified in the count action, and will appear in each flow which is part of this count. |

| | |
|---|---|
| Tally | Tally this packet. The attributes (or parts thereof) are specified by the mask attributes in the tally action specified by this rule's *Index*. The non-mask attributes of a tally action can be specified, and will appear in each flow which is part of this tally. |
| Aggregate | Aggregate this packet, i.e. count it in the aggregate action specified by this rule's *index.* All the attribute values of an aggregate flow are specified in the aggregate action. |
| Succeed | Succeed has two effects, depending on the rule's *Index* value.<br>If the *Index* value is zero, rule matching stops and a 'succeed' result is returned. This means the incoming packet will not be counted, i.e. it will be ignored.<br>If the *Index* value is not zero, it specifies another 'target' rule. The target rule's *Action* is executed immediately, as though that rule's test had just succeeded. This allows one rule to *Push* another's *Attribute* and *Mask* values. |
| Fail | Stop rule matching and return a 'fail' result. This allows the meter to interchange the source and destination attributes and retry the match. |
| PushTo | Save the current rule number 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 to construct *count* flows. |
| PopTo | Set the index of the next rule to be tested to this rule's *Index* and pop the topmost rule number from the pattern stack. |
| GoTo | Set the index of the next rule to be tested to this rule's *index.* |
| 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. |
| 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. |
| Assign | The attribute value specified by this rule's *Index* is assigned to the variable specified by this rule's *Attribute*. |

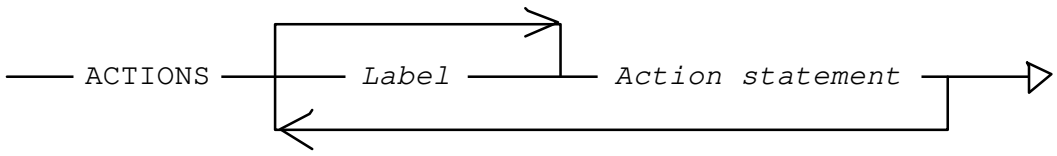
*Index*

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

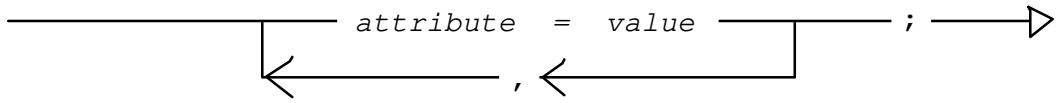
Rule and action numbers are 1-origin indexes to their corresponding tables. These numbers may be used directly, but it can be difficult to do this accurately. The simplest way to do this is to include the rule or action number in a comment attached to each statement, as indicated in the sample rule files.

Rules and actions 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 rule and action labels for consistency, flagging missing or duplicated labels.

### 5.2.3. Actions Section




**Action statement**




The Actions section specifies the action table for a rule set. It starts with the keyword ACTIONS, followed by a series of Action statements, one for each action.

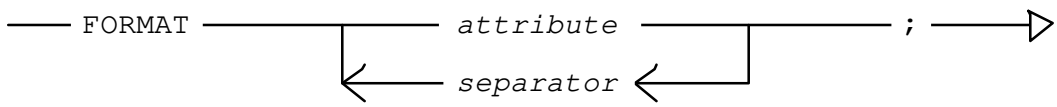
Each Action statement is a list of attribute values, separated by commas and terminated with a semicolon. Attribute values must be in the correct form for their attributes – these are explained in the 'Flow Attributes' section above.

Detail addresses are a special case. They have two-byte values, but may also be entered as single integers. Examples of this are given in the following sections.

Count and tally actions are used in the meter as the base of a hash table for all the flows in this count or tally. For a tally the masks used when searching the hash table are taken from the tally action, hence they will be the same for every flow in the tally. For a count the search masks are taken from the pattern stack (i.e. the rules which were matched for the current packet); the user should therefore ensure that every flow in a count uses the same masks.

### 5.2.4. Format Statement

**Format statement**

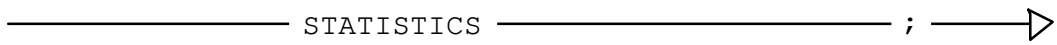



The Format statement specifies the format of rule data lines in a NeMaC Flow Data file.

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.

### 5.2.5. Statistics Statement

**Statistics statement**




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.

## 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 or action which is referred to by another rule. This is much easier than using explicit rule or action numbers.

- Check the rule file by using NeMaC's syntax check (-s and -v) 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

```
#  1705, Mon 10 Jan 94
#
#  Default rule file for NeTraMet (built in to the meter)
#
#  Nevil Brownlee,  Computer Centre,  The University of Auckland
#
SET 1
#
RULES
#
SourcePeerType & 255 = IP :         Aggregate,  1;  #   1
SourcePeerType & 255 = Novell :     Aggregate,  2;  #   2
SourcePeerType & 255 = DECnet :     Aggregate,  3;  #   3
SourcePeerType & 255 = EtherTalk :  Aggregate,  4;  #   4
SourcePeerType & 255 = CLNS :       Aggregate,  5;  #   5
SourcePeerType & 255 = Other :      Aggregate,  6;  #   6
```

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 six aggregate flows, one for each of the peer types which NeTraMet understands.

```
ACTIONS
#
SourcePeerType=IP, SourcePeerTypeMask=255,          #   1
   SourcePeerAddress=11.12.0.0, SourcePeerMask=255.255.0.0,
   DestPeerAddress=13.14.0.0, DestPeerMask=255.255.0.0;
SourcePeerType=Novell, SourcePeerTypeMask=255,      #   2
   SourcePeerAddress=21.22.0.0, SourcePeerMask=255.255.0.0,
   DestPeerAddress=23.24.0.0, DestPeerMask=255.255.0.0;
SourcePeerType=DECnet, SourcePeerTypeMask=255,      #   3
   SourcePeerAddress=31.32.0.0, SourcePeerMask=255.255.0.0,
   DestPeerAddress=33.34.0.0, DestPeerMask=255.255.0.0;
SourcePeerType=EtherTalk, SourcePeerTypeMask=255,   #   4
   SourcePeerAddress=41.42.0.0, SourcePeerMask=255.255.0.0,
   DestPeerAddress=43.44.0.0, DestPeerMask=255.255.0.0;
SourcePeerType=CLNS, SourcePeerTypeMask=255,        #   5
   SourcePeerAddress=51.52.0.0, SourcePeerMask=255.255.0.0,
   DestPeerAddress=53.54.0.0, DestPeerMask=255.255.0.0;
SourcePeerType=Other, SourcePeerTypeMask=255,       #   6
   SourcePeerAddress=61.62.0.0, SourcePeerMask=255.255.0.0,
   DestPeerAddress=63.64.0.0, DestPeerMask=255.255.0.0;
#
# end of file
```

The default action table provides aggregate actions as required by the default rules. Each of these specifies a set of values for flow attributes. These have been chosen to be values which would not occur in normal traffic flows – they simply serve to identify the aggregates.

### 5.3.2. rules.sample

```
#  1705, Mon 13 Sep 93
#
#  Rule specification file to tally IP net <-> IP net,
#     tally DECnet and Novell and aggregate EtherTalk
#
#  Nevil Brownlee,  Computer Centre,  The University of Auckland
#
SET 2  #  Rule + action set number 2
#
RULES
#
SourcePeerType & 255 = DECnet :         Tally,  9;  #   1
SourcePeerType & 255 = IP :            Pushto,  8;  #   2
SourcePeerType & 255 = EtherTalk :  Aggregate,  7;  #   3
```

```
SourcePeerType & 255 = Novell :          Tally,  8;  #   4
Null & 0 = 0 :                          Succeed,  0;  #   5  Ignore
#
Null & 0 = 0 :                          Succeed,  0;  #   6
Null & 0 = 0 :                          Succeed,  0;  #   7
```

The first part of this rule table determines the peer protocol type. DECnet and Novell packets are tallied by actions 9 and 8, and EtherTalk packets are aggregated using action 7. IP packets cause the meter to save rule 2's index on the pattern stack then test rule 8. Rules 6 and 7 are never executed; they merely provide a gap in the rule table for possible later use.

Rule five, above, tests the *Null* attribute. This is a special case which always succeeds. The rule action is *Succeed,* which means that any other packets will be ignored.

```
SourcePeerAddress & 192.0.0.0 = 128.0.0.0 :  Pushto, 13;  #   8  Low B
SourcePeerAddress & 192.0.0.0 = 192.0.0.0 :  Pushto, 16;  #   9  Low C
SourcePeerAddress & 192.0.0.0 = 64.0.0.0 :   Pushto, 18;  #  10  Low A
SourcePeerAddress & 192.0.0.0 = 0.0.0.0 :    Pushto, 18;  #  11  Low A
Null & 0 = 0 :                                 Fail,  0;  #  12
#
DestPeerAddress & 192.0.0.0 = 128.0.0.0 :     Tally,  1;  #  13  High B: B-B
DestPeerAddress & 192.0.0.0 = 192.0.0.0 :     Tally,  2;  #  14  High C: B-C
Null & 0 = 0 :                                 Fail,  0;  #  15
#
DestPeerAddress & 192.0.0.0 = 192.0.0.0 :     Tally,  3;  #  16  High C: C-C
Null & 0 = 0 :                                 Fail,  0;  #  17
#
DestPeerAddress & 192.0.0.0 = 128.0.0.0 :    Tally,   4;  #  18  High B: A-B
DestPeerAddress & 192.0.0.0 = 192.0.0.0 :    Tally,   5;  #  19  High C: A-C
DestPeerAddress & 192.0.0.0 = 64.0.0.0 :     Tally,   6;  #  20  High A: A-A
DestPeerAddress & 192.0.0.0 = 0.0.0.0 :      Tally,   6;  #  21  High A: A-A
Null & 0 = 0 :                                 Fail,  0;  #  22
#
```

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. The packet is then tallied in one of six tallies for each of the possible pairs of address class.

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.

```
ACTIONS
#
SourcePeerType=IP, SourcePeerMask=255.255.0.0,
                   DestPeerMask=255.255.0.0;            # 1  B-B
SourcePeerType=IP, SourcePeerMask=255.255.0.0,
                   DestPeerMask=255.255.255.0;          # 2  B-C
SourcePeerType=IP, SourcePeerMask=255.255.255.0,
                   DestPeerMask=255.255.255.0;          # 3  C-C
SourcePeerType=IP, SourcePeerMask=255.0.0.0,
                   DestPeerMask=255.255.0.0;            # 4  A-B
SourcePeerType=IP, SourcePeerMask=255.0.0.0,
                   DestPeerMask=255.255.255.0;          # 5  A-C
SourcePeerType=IP, SourcePeerMask=255.0.0.0,
                   DestPeerMask=255.0.0.0;              # 6  A-A
```

The first six actions are tallies for each of the possible combinations of IP address class. Each has the appropriate peer address masks set, telling the meter which parts of those attributes are to be tallied. *SourcePeerType* is set to IP; without this it would not be possible to determine the peer address type when the flow data was collected.

```
SourcePeerType=EtherTalk,
   SourcePeerAddress=1.0.0.0, SourcePeerMask=255.0.0.0,
   DestPeerAddress=1.0.0.0, DestPeerMask=255.0.0.0;     # 7  Aggregate
```

```
#
SourcePeerType=Novell, SourcePeerMask=255.255.255.255,
                       DestPeerMask=255.255.255.255;    # 8  Tally all
SourcePeerType=DECnet, SourcePeerMask=255.255.255.0,
                       DestPeerMask=255.255.255.0;      # 9  Tally all
```

The remaining actions aggreagate EtherTalk and tally Novell and DECnet packets. The tallies use the entire peer address in each case. This instructs the meter to create flows for every possible pair of peer addresses. The network segment which this rule set was written for has very little EtherTalk, Novell or DECnet traffic.

```
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

```
#  1605, Fri 24 Sep 93
#
#  Rule specification file to tally traffic to/from ccr1
#
#  Nevil Brownlee,  Computer Centre,  The University of Auckland
#
SET 6  #  Rule + action set number 6
#
RULES
#
DestAdjacentAddress & ff-ff-ff-ff-ff-ff = aa-00-04-00-f4-ed :  #  ccr1
    Pushto,  3;  #   1
Null & 0 = 0 :         Fail,  0;  #   2  Try again
#
SourcePeerType,       255,           IP,        Pushto,  5;  #   3
Null,                   0,            0,       Succeed,  0;  #   4  Ignore
#
SourcePeerAddress,  255.255.0.0,  130.216.0.0,  Pushto,  8;  #   5  Auckland
SourcePeerAddress,  192.0.0.0,    192.0.0.0,    Pushto,  8;  #   6  Low C
Null,                 0,            0,          Pushto, 11;  #   7  Low B
```

This rule set was intended for metering traffic through a gateway ethernet, which has a router named *ccr1* providing our connection to the Internet. Rule 1 above tests each packet's *DestAdjacentAddress* to see whether its destination is on the far side of *ccr1.* If it is, rule 5 will be tested next, otherwise rule two is tested, Rule two fails, 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

```
#  1630, Thu 9 Dec 93
#
#  Rule specification file to tally broadcast packets
#
#  Nevil Brownlee,  Computer Centre,  University of Auckland
#
SET 5  # Rule + action set number 5
#
RULES
#
DestAdjacentAddress & FF-FF-FF-FF-FF-FF = FF-FF-FF-FF-FF-FF: Tally, broadcasts;
Null & 0 = 0 :  Fail, 0;  #  Try other direction
```

This rule set looks for ethernet broadcast packets, by testing their *DestAdjacentAddress*. Broadcasts are tallied using the action labelled 'broadcasts' (see below).

The second rule is very important, and is needed because of the way tallies are handled. Consider an incoming packet. If rule 1 matches it is a broadcast packet and is tallied. If this (broadcast) flow is already in the tally it is counted, and the job is done. If it's not, Tally 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 tally which caused the first failure so the match is tried a third time, this time forcing the tally to add the flow.

```
ACTIONS
#
broadcasts:
SourcePeerTypeMask=255,
   SourceAdjacentMask=FF-FF-FF-FF-FF-FF, DestAdjacentMask=FF-FF-FF-FF-FF-FF,
   SourcePeerMask=255.255.255.255, DestPeerMask=255.255.255.255,
   SourceDetailTypeMask=255,
   SourceDetailMask=255.255, DestDetailMask=255.255;
#
FORMAT FlowRuleSet FlowIndex FirstTime "  "
   SourceAdjacentAddress DestAdjacentAddress "  "
   SourcePeerType
   SourcePeerAddress DestPeerAddress "  "
   SourceDetailType SourceDetailAddress DestDetailAddress "  "
   ToPDUs FromPDUs "  " ToOctets FromOctets;
#
STATISTICS  #  Collect meter statistics
#
# end of file
```

There is only one action, labelled 'broadcasts,' which collects all the details of each broadcast flow. Since the *SourcePeerTypeMask* is 255, broadcast flows will be collected for all the protocols the meter recognises. If we only wanted IP broadcasts we would need another rule to test *SourcePeerType* and ignore non-IP packets.

Notice that because we have used non-zero masks in the tally, the corresponding fields are saved in the flows. This demonstrates the usefulness of tallies for collecting a large amount of detail without explicitly testing each required attribute and using a Count action to build the flow.

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.manage

```
#  1625, Fri 7 Jan 94
#
#  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 8  # Rule + action set number
#
RULES
#
SourcePeerType & 255 = IP :   Pushto,  IP_pkt;
Null & 0 = 0 :               Succeed,  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 = 0 :  Assign, SourcePeerAddress;
Null & 0 = 0:  Gosub, Auckland_nets;
Null & 0 = 0:    Goto, c_pkt;  # 1 Dept -> dept-local
Null & 0 = 0 :  Tally, t_bad;  # 2 UA, not in list of OK subnets -> UA-local
Null & 0 = 0:     Succeed, 0;  # 3 Local -> Ignore local-local
```

```
Null & 0 = 0 :       Fail, 0;  # 4 Not UA or local -> Want local as source
#
dest_UA:
SourcePeerAddress & 255.255.0.0 = 130.216.0.0 :  Succeed, 0; # Ignore UA-UA
Null & 0 = 0 :   Fail, 0;  #  Want Auckland as source
#
IP_pkt:
DestPeerAddress & 255.255.0.0 = 130.216.0.0 :  Pushto, dest_UA; # Auckland
#
v1 & 0 = 0 :  Assign, DestPeerAddress;
Null & 0 = 0:  Gosub, Auckland_nets;
Null & 0 = 0:          Succeed, 0;        # 1 dest UA department
Null & 0 = 0:          Succeed, 0;        # 2 dest UA
Null & 0 = 0:    Goto, dest_local;        # 3 dest Local
#
v1 & 0 = 0 :  Assign, SourcePeerAddress;  # 4 dest NZ or world
Null & 0 = 0:  Gosub, Auckland_nets;
Null & 0 = 0:    Goto, src_dept;          # 1 source Dept
Null & 0 = 0 :     Tally, t_bad;          # 2 source UA, not an OK subnet
Null & 0 = 0:   Goto, src_local;          # 3 source Local
Null & 0 = 0 :     Tally, 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 = 0 :  Assign, DestPeerAddress;
Null & 0 = 0:  Gosub, Tuia_proximal;
DestPeerAddress & 255.255.0 = 130.216.0 : Succeed, 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_Tuia; # 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 = 0 :  Assign, DestPeerAddress;
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 & 255.255 = 140.200 : Pushto, c_Tuia; # 17 Tuia
DestPeerAddress & 255 = 253.0.0 : Pushto, c_pkt; # 18 Unconnected
DestPeerAddress & 255 = 254.0.0 : Pushto, c_pkt; # 19 World
#
c_Tuia:
DestPeerAddress & 255.255.255.255 = 140.200.240.8 : Tally, t_NL_stun;
```

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:
SourceDetailType & 255 = icmp :           Count, c_det_only;    # ICMP
#
s_news:
SourceDetailAddress & 255.255 = nntp :    Count, c_det_source;  # News
DestDetailAddress & 255.255 = nntp :      Succeed, s_news;
s_smtp:
SourceDetailAddress & 255.255 = smtp :    Count, c_det_source;  # Mail
DestDetailAddress & 255.255 = smtp :      Succeed, s_smtp;
s_domain:
SourceDetailAddress & 255.255 = domain :  Count, c_det_source;  # DNS
DestDetailAddress & 255.255 = domain :    Succeed, s_domain;
s_telnet:
SourceDetailAddress & 255.255 = telnet :  Count, c_det_source;  # Telnet
DestDetailAddress & 255.255 = telnet :    Succeed, s_telnet;
s_ftp_ctrl:
SourceDetailAddress & 255.255 = ftp :     Count, c_det_source;  # FTP control
DestDetailAddress & 255.255 = ftp :       Succeed, s_ftp_ctrl;
#
DestDetailAddress & 255.255 = ftpdata :   Count, c_det_dest;    # Local FTP
SourceDetailAddress & 255.255 = ftpdata : Count, c_det_source;  # Remote FTP
#
SourceDetailType & 255 = tcp :            Count, c_det_only;    # Other TCP
SourceDetailType & 255 = udp :            Count, c_det_only;    # Other UDP
#
Null & 0 = 0 :  Tally, t_bad;     #  End of packet testing
```

The rules starting at c_pkt test the detail 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; detail 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    : Pushto, A_dept;  #  Computer Centre
v1 & 255.255.255.0 = 130.216.3.0    : Pushto, A_dept;  #  Auckland DMZ
v1 & 255.255.255.0 = 130.216.5.0    : Pushto, A_dept;  #  Engineering Science
v1 & 255.255.255.0 = 130.216.7.0    : Pushto, A_dept;  #  Physics
v1 & 255.255.255.0 = 130.216.11.0   : Pushto, A_dept;  #  Medical School
v1 & 255.255.255.0 = 130.216.12.0   : Pushto, A_dept;  #  Pharmacology
v1 & 255.255.255.0 = 130.216.14.0   : Pushto, A_dept;  #  Commerce
v1 & 255.255.255.0 = 130.216.15.0   : Pushto, A_dept;  #  Mathematics
v1 & 255.255.255.0 = 130.216.21.0   : Pushto, A_dept;  #  Chemistry
v1 & 255.255.255.0 = 130.216.26.0   : Pushto, A_dept;  #  School Bio Sciences
```

```
    v1 & 255.255.255.0 = 130.216.33.0   : Pushto, A_dept;  #  Computer Science
    v1 & 255.255.255.0 = 130.216.34.0   : Pushto, A_dept;  #  Computer Science
    v1 & 255.255.255.0 = 130.216.73.0   : Pushto, A_dept;  #  Law
    #
    v1 & 255.255.255.0 = 192.156.165.0  : Pushto, A_local; #  DECUSLINK
    v1 & 255.255.255.0 = 192.251.230.0  : Pushto, A_local; #  CLEARFIELD
    v1 & 255.255.255.0 = 202.12.104.0   : Pushto, A_local; #  DSE
    v1 & 255.255.255.0 = 202.12.105.0   : Pushto, A_local; #  FPNET
    v1 & 255.255.255.0 = 202.14.100.0   : Pushto, A_local; #  STATUS
    v1 & 255.255.255.0 = 202.14.102.0   : Pushto, A_local; #  KCBBS
    v1 & 255.255.255.0 = 202.14.216.0   : Pushto, A_local; #  MANUKAU
    v1 & 255.255.255.0 = 202.14.217.0   : Pushto, A_local; #  MALEFICARUM
    v1 & 255.255.255.0 = 202.14.252.0   : Pushto, A_local; #  NETBLK-CRAYCOM
    v1 & 255.255.255.0 = 202.14.253.0   : Pushto, A_local; #  NETBLK-CRAYCOM
    v1 & 255.255.255.0 = 202.14.254.0   : Pushto, A_local; #  NETBLK-CRAYCOM
    #
    v1 & 255.255.0.0 = 156.62.0.0       : Pushto, 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
    #
    #   IP          Name            TuiaNet connection site
    #
    # 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

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

    v1 & 255.255.0.0 = 140.200.0.0      : Return, 17; #  KAWAIHIKO
    #
    #  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

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

    Null & 0 = 0                        : Return, 19; #  Not an NZ net
    #
```

```
ACTIONS
#
t_bad:
SourcePeerTypeMask=255,              # Tally unexpected packets
   SourcePeerMask=255.255.255.255, DestPeerMask=255.255.255.255,
   SourceDetailTypeMask=255,
   SourceDetailMask=255.255, DestDetailMask=255.255;
#
t_NL_stun:
SourcePeerTypeMask=255,              # Tally STUN packets to NZBN
   SourcePeerMask=255.255.255.255, DestPeerMask=255.255.255.255,
   SourceDetailTypeMask=255;
#
c_det_only:
SourcePeerType=ip,                   # Count ICMP, Other TCP, Other UDP
   SourcePeerMask=255, DestPeerMask=255,
   SourceDetailTypeMask=255;
#
c_det_source:
SourcePeerType=ip,                   # Count DNS, News, Remote FTP
   SourcePeerMask=255, DestPeerMask=255,
   SourceDetailMask=255.255;
#
c_det_dest:
SourcePeerType=ip,                   # Count Local FTP
   SourcePeerMask=255, DestPeerMask=255,

 DestDetailMask=255.255;
```

The counts above have masks set to indicate how they are used. These are only comments though, because Count uses the masks (and values) from the pattern stack.

```
FORMAT FlowRuleSet FlowIndex FirstTime "  "
   SourcePeerType
   SourcePeerAddress DestPeerAddress "  "
   SourceDetailType SourceDetailAddress DestDetailAddress "  "
   ToPDUs FromPDUs "  " ToOctets FromOctets;
#
STATISTICS  #  Collect meter statistics
#
# end of file
```

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

### 5.3.6. rules.lan

```
#  1015, Mon 4 Oct 93
#
#  Rule specification file to tally Local Area Network traffic
#
#  Nevil Brownlee,  Computer Centre,  The University of Auckland
#
SET 4  # Rule + action set number
#
RULES
#
SourcePeerType & 255 = IP :             Tally,  1;  #   1
SourcePeerType & 255 = DECnet :        Pushto,  6;  #   2
SourcePeerType & 255 = EtherTalk :     Pushto, 11;  #   3
SourcePeerType & 255 = Novell :         Tally,  6;  #   4
Null & 0 = 0 :                         Succeed,  0;  #   5  Ignore
#
SourceDetailType & 255 = 38 : Tally,  2;  #   6  Tally DECnet data by host
SourceDetailType & 255 =  6 : Tally,  2;  #   7
SourceDetailType & 255 = 46 : Tally,  2;  #   8
SourceDetailType & 255 = 14 : Tally,  2;  #   9
Null & 0 = 0 :                Tally,  3;  #  10  Tally rest of DECnet by
detail
#
SourceDetailType & 255 =  3 : Tally,  4;  #  11  Tally AT data by host
```

```
Null & 0 = 0 :                Tally,  5;  #  12  Tally rest of AT by detail
#
ACTIONS
#
SourcePeerType=IP,                      #  1  Tally IP by subnet and port
   SourcePeerMask=255.255.255.0, DestPeerMask=255.255.255.0,
   SourceDetailTypeMask=255, SourceDetailMask=255.255, DestDetailMask=255.255;
#
SourcePeerType=DECnet,                  #  2 Tally DECnet data by host
   SourcePeerMask=255.255.255.0, DestPeerMask=255.255.255.0,
   SourceDetailTypeMask=255;
SourcePeerType=DECnet,                  #  3 Tally DECnet by protocol
   SourceDetailTypeMask=255;
#
SourcePeerType=EtherTalk,               #  4 Tally Ethertalk data by host
   SourcePeerMask=255.255.255.0, DestPeerMask=255.255.255.0,
   SourceDetailTypeMask=255;
SourcePeerType=EtherTalk,               #  5 Tally Ethertalk by DDP type
   SourceDetailTypeMask=255;
#
SourcePeerType=Novell,                  #  6  Tally IPX by network nbr and port
   SourcePeerMask=255.255.255.255, DestPeerMask=255.255.255.255,
   SourceDetailTypeMask=255,
   SourceDetailMask=255.255, DestDetailMask=255.255;
#
STATISTICS
#
#FORMAT FlowRuleSet FlowIndex FirstTime "  "
   SourcePeerType SourcePeerAddress DestPeerAddress "  "
   SourceDetailType SourceDetailAddress DestDetailAddress "  "
   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 detail 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' Manual

## 6.1. Overview of NeTraMet

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 nameserver). 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.

The options are:

**-c nnn** Specifies the required collection interval in seconds. If nnn is zero the flow data will be collected once, then NeMaC will exit; this can be useful, for example for changing the rule set back to its default value of 1 if no flows are to be collected.

**-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.

**-l** Requests NeMaC to list the rule file(s) as they are processed.

**-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.

**-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.

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.

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 cheak. Output from this operation is directed to a file called 'syntax.special' for later inspection.

### 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:

**-g sss** Specifies the meter's garbage collection interval.

**-h pp** Sets NeTraMet's *HighWaterMark* as a percentage of the available flow space. Default value is 85%.

**-i sss** Specifies the meter's *InactivityTimeout* interval in seconds. Default value is 600 seconds.

**-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.

## 7. NeTraMet Users' Manual

### 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.

The options are:

**-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 any of the read communities.

**-r rsc** Specifies a read SNMP community for NeTraMet. Up to four read communities may be specified.

**-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 a background process under SunOS you *must* disable the keyboard.

**-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.

**-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.

**-p nnn** Sets the size of NeTraMet's buffer for incoming packet headers. The default size is 1024 packet headers.

## 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 also 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 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:

**b** Display 'bad packets' counts

**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

**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 are as follows:

```
Av pkt/s 90, av pkt backlog 1
Max pkt/s 154, max pkt backlog 1
Idle time av 98.2, min 96.5 %
247 flows in use (max 4000)
13 flows recovered (GC: 5  4 16)
2.3 rules/pkt, 1.2 tallies/pkt
1.8 compares/tally
512 hash slots, 83 in use
```

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 In Use' means flows which currently hold valid flow data. Flows normally remain in use until they become inactive and their data has been collected. Once they 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 flows first.

'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 three 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 parameters are the number of 'in use' and 'idle' flows tested by the garbage collector each time it is invoked. These cannot be changed by the user.

'Rules per packet' and 'Tallies per Packet' show how many rules were tested and how many tally actions were performed for each packet. Tallies are implemented by hash tables; the number of 'Compares per Tally' gives an indication of how long the hash chains have become.

'Number of Hash Slots' is the maximum number of entries in the tally hash tables. The number of hash slots in use indicates how effectively the hashing function is spreading the incoming packets across the hash tables.

## 7.5. 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.

### 7.6. Sample AUTOEXEC.BAT file

```
wd8003e  0x60  5 0x300 0xD800
NeTraMet  -r NOC 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 Version 2.0, NeTraMet can only handle a single packet driver.

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

### 7.7. Differences between PC and SunOS versions of NeTraMet

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

The SunOS version is intended to run as a background process on a Unix 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 SunOS 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 SunOS. Packet backlog and processor utilisation statistics are not implemented for SunOS.

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 SunOS 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). The SunOS limit is the fact that *FlowIndexes* are 16-bit values, giving an upper limit of 65,534 flows.

## 8. NeTraMet Distribution

### 8.1. Copyright Statement

NeTraMet is free software, distributed under the terms of the GNU Emacs general public license. A copy of this is provided with the NeTraMet software distribution files.

### 8.2. Distribution Files

There are two NeTraMet distribution files, containing the following:

*NeTraMet.tar.Z*

NeTraMet documentation, including example rule files and the Meter Services MIB
Source code for CMU SNMP, NeTraMet and NeMaC
Make files for NeTraMet and NeMaC on SunOS
Make files for NeMaC on Irix and Ultrix
NeTraMet.exe file for DOS

*NeTraMet.zip*

*S*ource code for Waterloo TCP, CMU SNMP and NeTraMet
Turbo Make files for these

## 9. Installation

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

| | |
|---|---|
| `./documentation` | |
| `/NeTraMet` | NeTraMet documentation |
| `/examples` | rule files |
| `/snmp` | CMU SNMP documentation |
| `./mib` | Accounting MIB |
| `./src` | |
| `/snmplib` | CMU SNMP library source |
| `/apps` | CMU SNMP applications source |
| `/meter` | NeTraMet source |
| `/manager` | NeMaC source |
| `./sun/` | Make files for SunOS |
| `./sg/` | Make files for Irix `NeTraMet.` |
| `./ultrix/` | Make files for Ultrix `NeTraMet.` |
| `./pc/` | EXE and WATTCP.CFG files for DOS |

*On a Unix system (general notes)*

Each Unix system has a directory with *make* files for that system. Examine the *make* files for your system – you may need to change some of the options. There are two compile-time options for NeTraMet, as follows:

| | | |
|---|---|---|
| CLNS | Set 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 |

The SunOS *make* files use acc, Sun's ANSII C compiler. You may wish to use gcc, the Gnu C compiler, instead. To do this you can either change the CC variable in the make files, or you can set the CC environment variable (`setenv CC gcc`) and run make with the -e option (`make -e`).

*On a SunOS machine:*

*cd* to the sun/snmplib directory and run the *make* file; this will create the SNMP library.

*cd* to the sun/apps directory and run the *make* file; this will create a minimal set of SNMP application programs..

*cd* to the sun/meter directory and run the *make* file; this will create NeTraMet.

*cd* to the sun/manager directory and run the *make* file; this will create NeMaC.

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 an Irix or Ultrix machine:*

Proceed as above. The only differences are that the Irix make *files* are in the /sg directory instead of the /sun directory, and that NeTraMet is not implemented yet for Irix.

*On a PC:*

If you only want to use NeTraMet, copy NeTraMet.exe and WATTCP.CFG from your Unix machine to the PC. Edit WATTCP.CFG to specify your meter's IP configuration. Start NeTraMet.

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

| | |
|---|---|
| `/elib` | Waterloo TCP intrinsics library source |
| `/src` | Waterloo TCP library source |
| `/apps` | Waterloo TCP application programs |
| `/snmp` | CMU SNMP library source |
| `/acct` | NeTraMet source |
| `/lib` | NeTraMet libraries |

To create NeTraMet, run Turbo Make in each of the elib, src, snmp and acct directories in turn. This will create NeTraMet.exe in the acct directory.

## 10. NeTraMet's Future

This document desccribes 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.

Discussion about the Internet Accounting Architecture will continue on the Working Group's mailing list, accounting-wg@wugate.wustl.edu. If you are interested in network traffic metering please join this list by sending a request to accounting-wg-request @wugate.wustl.edu. Topics needing further work include

- Improvements or modifications to the Meter Services MIB in the light of users' experience.
- Development of a language for describing metering requirements, and a compiler to test these for consistency and translate them into rules.
- Development of standards for flow data files and their management.

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

- Performance of various configurations of PC and SunOS 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.
- Rule files. How hard or easy did you find it to create rule files for your metering requirements? Were there things you wanted to do but couldn't? Did you discover any particularly elegant sets of rules for metering your network?

Please post 'experience' reports to the Internet Accounting Mailing List (address above).

## 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 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

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

Scott Marcus (BBN)
Tran Phan Anh (University of Kansas)

## 11. References

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

"RFC 1060: Assigned Numbers," J. Reynolds, J. Postel, March 1990

"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