Measuring Network Utilisation at the University of Melbourne

Douglas Ray

Information Technology Services
University of Melbourne

ABSTRACT

Having decided what needs measuring and how to catch it, there is still many a slip between the initial configuration of applications and the final product, the network utilisation report.

Our task was measurement and characterisation of national and international traffic with areas of the university. We discuss how the functionality of NNStat is applied to this task, and the additional features required to incorporate NNStat within a reliable report generator for network statistics.

Some preliminary performance measurements are presented, and their implications considered.

This paper was presented at the AARNet Networkshop during December 1993 (Melbourne, Australia).

Douglas Ray has worked within the ITS Networks and Communications Group at the University of Melbourne for the past four years. He has regularly given presentations at Networkshop and AUUG conferences. He can be reached at either doug@unimelb.edu.au or doug@munnari.oz.au

---

0 Context

This paper explores a specific project in network usage monitoring, currently in use at the University of Melbourne. The project was partly prompted by the perception that at some stage charging for external network traffic may become necessary within the University.

Usage is a fiddly thing to define, because unless you do protocol emulation on every connection you can't always determine which side of a connection initiated a data transfer. This paper does not address that problem.

To date, the possible mode of charge has been assumed to be a measure of volume usage, possibly weighted by type-of-service. Byte and packet measures have been derived, and per-service figures are kept.

Within this limited frame there are many possible strategies. Links have fixed bandwidth, routers have upper limits on packet switching, so depending on the degree of saturation, and whether a route is saturated at link or at gateway, different weightings of byte and packet measures may be applicable – *eg*, straight byte-based accounting for saturated links, and a combination of byte and packet accounting for non-saturated links and router bottlenecks.

However, this begs the question of what is trying to be achieved in charging. The rationale behind the above example is that charges should directly reflect the cost of the infrastructure (links and routers) being used. Further, the implication is that charging is a means of recouping costs. This ignores the utility of charging in managing usage patterns.

If you're trying to curtail usage of saturated links, there's not much point putting a financial premium on their use unless the person being charged knows when they are suffering the premium. To effect such behaviour it is more effective – not to mention much simpler – to specify peak usage periods, and charge a premium on usage in those periods.

These, and other strategies, and references, can be found in Roger Clarke's paper for this conference. My concern is not so much the charging strategy as the method of measuring usage. The foregoing discussion serves to emphasise that *usage* has as many meanings as one chooses to define. The definition we're after is the one which saves us money.

1 Task

The task I was presented with was to measure and characterise types of traffic between parts of the University and the rest of Victoria, greater Australia, and overseas. (By *characterisation* of traffic I mean measuring various types of services used – telnet, mail, ftp, *etc*.)

It quickly became apparent that it would be useful to log a fourth class separately: traffic with the central AARNet servers.

Only IP traffic is considered. This forms the bulk of traffic exiting the campus network.

"Parts of the University" was clarified to be subnets. This may need to be refined in the light of possible charging applications. If one assumed we wanted to generate bills at a departmental or faculty level there are a few problems we must solve.

A) Departments with mixed subnets

This is simple – add the subnets – and we probably need to do this anyway. However, there is a substantial chunk of work in adding the further level of abstraction, department, on top of the existing subnet-based processing.

B) Subnets with mixed departments

This is the killer. The best solution is not to mix departments on a subnet, but given our current subnet usage that would be impractical. Also, there are some departments that naturally have hosts dotted all over campus; *eg*, the library. All solutions, other than segregation of departments at the subnet level, require changing the granularity of local peer from subnet to host, and collating the host info by department. (For a note on the practicality of this, see section 7). Even if we presume that performance limits would allow this, we also need procedures which enable automatic updates of our stats monitor config whenever new hosts appear, move between departments, or when departments move between subnets.

One solution is to collate IP's by DNS zone name. This presumes that the zone names in the DNS adequately reflect departmental structure for billing purposes.

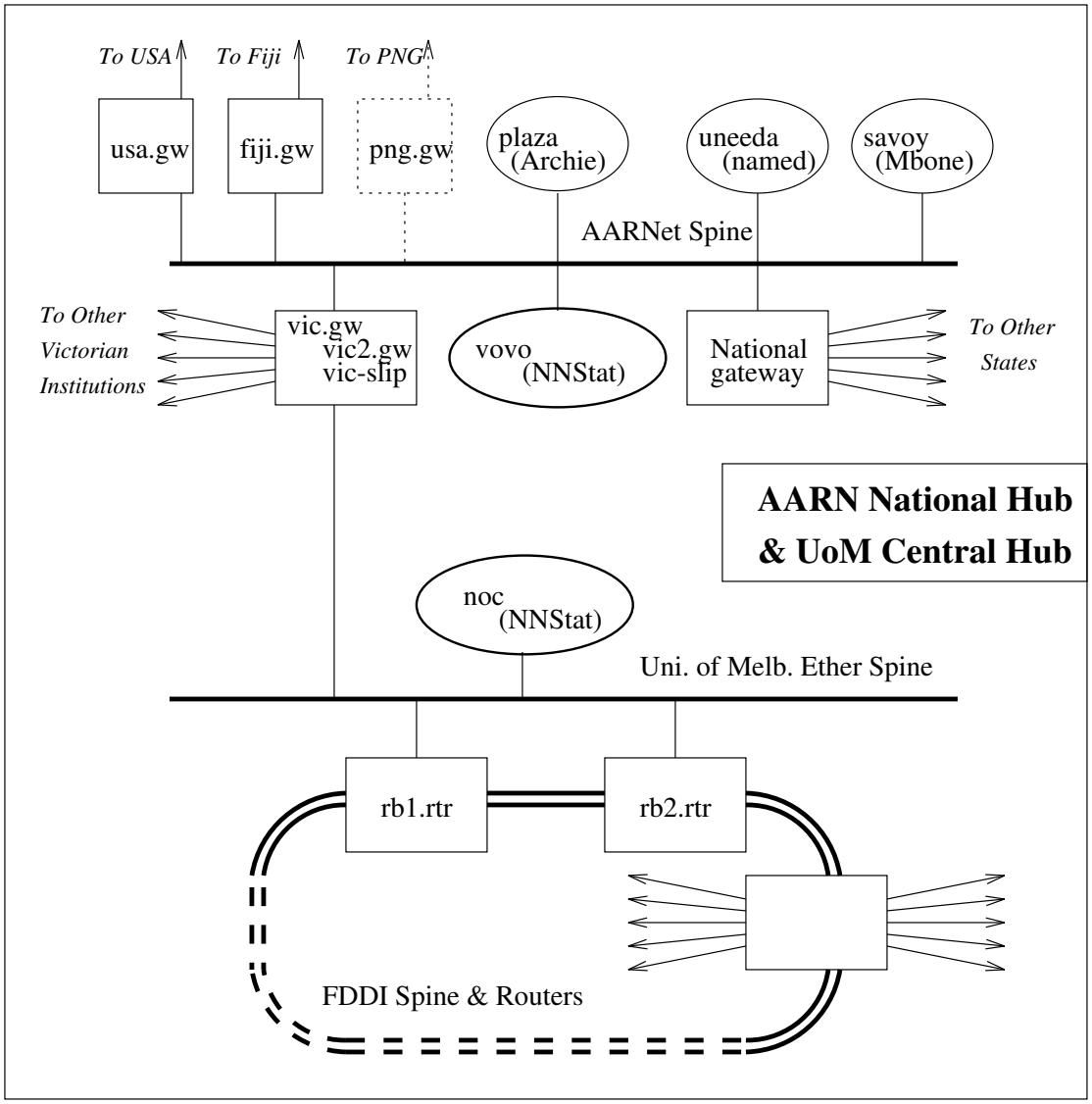
If they don't, then a separate table of IP number/department pairs (or hostname/department pairs, with the IP number being found by DNS query). This assumes that the list will be maintained – which requires extra work on updates – or that the DNS files will be automatically regenerated from the list. It also assumes that those departments running their own nameservers remember to forward information of new hosts to us... information for which they will be billed.

C) Subnets with shared resources

As an example, we have various hosts in lecture theatres, for which someone would have to work out a billing system.

2 Network Topology

We measure traffic on the following network.



The key points of this topology are as follows. The AARNet spine has

- servers
- a link to Internet via the USA
- links to other states via the national router
- links to Victorian institutions via the Victorian router

One of these latter links connects to the University of Melbourne's ethernet spine. The UoM ether spine

supports two routers making redundant connections to the main FDDI ring around campus. (There are a further 6 cisco routers on the FDDI ring.)

The University is primarily a class B network using a subnet mask of 255.255.255.0. Only a handful of subnet numbers are not in use. Variable length subnet masks have recently been enabled, but the implications of this for our stats reporting methodology are not discussed here.

3 Sources of Info (on campus)

3.1.1 Cisco Routers

The cisco routers can be configured for IP logging. This gives IP source and destination pairs, with packet and byte counts. The drawbacks of this approach are:

- can't distinguish type-of-service.
  (for this we'd need the port numbers)
- no control of granularity: must collate info for each host-host
  pair (not to mention download the info across the network)
- load on the routers
  (IP logging imposes a substantial CPU overhead on ciscos)
- doesn't give geographic location of peer.

3.1.2 Stats Servers

Dedicated stats servers have several advantages over router accounting. Using generic packet filtering, we can select by

- fields in the ethernet header
- fields in the IP header
- derived info ("virtual fields": *eg*, network and subnet)

This makes it easy to differentiate type-of-service. Dedicated stats servers impose little or no performance overhead on network load. (We find this functionality in the NSFnet software package, NNStat, which is used on the AARNet stats server **vovo** and on our stats server.)

If we use stats servers, where do we put them?

3.1.2.1 Departmental Stats Servers

A dedicated stats server on each subnet would have the advantage of measuring real traffic on the subnets, rather than just traffic crossing a cisco interface.

However, though this could be useful for maintenance and load prediction, it is not part of the task under discussion. It is also particularly expensive, both in equipment and configuration time.

3.1.2.2 Central Stats Server

A single stats server on the UoM ether spine would see each packet entering or exiting the University. Results can be processed locally on the server rather than being shipped across the network, and the presence of the server doesn't affect network performance. This is the approach we've taken.

3.2 Geographic Info

Remember, our task is to separate Victorian, Australian and international traffic. How do we get the geographic location of the peer?

Traffic crossing the UoM ether spine will have an ethernet address (either source or destination) of the victorian gateway. This differentiates the traffic we want from any other stuff, but still doesn't give us the geographic info we need.

The only other source of information at the UoM ether spine is the IP address of the peer. That means we need a list of network numbers and their geographic location. What range of network numbers do we

wish to cover? We don't need (or want) every network in the world. If we had, separately, all Victorian networks and all other Australian networks, then anything else can be assumed to be overseas. (But all we really require are those Victorian and Australian nets that talk with us.)

We might consider traceroute (but not for long :). Even with a maximum hop count of two, using traceroute to determine the location of IP numbers would be an excellent way of consuming system resources. At one extreme one traces every address not currently known, recording Victorian or Australian networks in lists – then each overseas address generates a trace whenever it occurs. T'other extreme, we maintain a list of overseas addresses as well... but that list would get rather large. Okay, well, we can *cache* the overseas addresses... but one starts to sense it could be worth looking for a simpler solution.

There's a file which contains details of all Australian IP networks, on **munnari.oz.au**: **netinfo/status**. Unfortunately, although the network numbers and network names are kept up to date, the geographic information is not complete.

Traffic crossing the AARNet spine has an ethernet peer address of the Victorian gateway, the national gateway, or the usa gateway (for simplicity we'll ignore the fiji gateway). This is where we can distinguish our target geographic classes. Luckily, the AARNet stats server (vovo) already does this, so we can get lists of Australian and Victorian IP networks from there. We download the files listing Victorian networks and their Australian network connection peers. This is an adequate approximation of the required "Victorian and Australian nets that talk with us", ignoring any Victorian nets that talk exclusively with us.

4 Algorithm and Constraints

4.1 Algorithm

Here we discuss the algorithm used for sorting and collating the traffic.

As mentioned above, the common factor in all traffic we're interested in is that, when crossing the UoM ether spine, it has either a source or destination ethernet address of the victorian gateway. This gives us the outer tests of our algorithm. Given the lists of network numbers we can easily derive the basic structure, which uses eight sets of tests to record traffic in and out of each of our four target locations:

```
if ether source = vic.gw
then {
        if IP source network is in VICnets
                record in incoming-traffic-from-vic
        else if IP source subnet is in AARNspine
                record in incoming-traffic-from-aarn
        else if IP source network is in AUSnets
                record in incoming-traffic-from-aus
        else
                record in incoming-traffic-from-os
}
else if ether destination = vic.gw
then {
        if IP destination network is in VICnets
                record in outgoing-traffic-to-vic
        else if IP destination subnet is in AARNspine
                record in outgoing-traffic-to-aarn
        else if IP destination network is in AUSnets
                record in outgoing-traffic-to-aus
        else
                record in outgoing-traffic-to-os
}
```

From here, the pseudocode "record" must be expanded to a block of tests which logs at the particular level of granularity required. (This will be discussed later: see section 6.2.1.) In our context this means a series of tests to distinguish port information, and special tests for particular servers and subnets.

4.2 Constraints of the Algorithm

Traffic with addresses not registered in our lists of Victorian and Australian networks will be wrongly attributed to overseas traffic. Also, networks are dynamic. More arrive continually, on something between a daily and weekly basis. To address these points we need timely updates of the network lists.

Unavoidably, we also need to rebuild the config of our network monitoring software on a daily basis, to include updated network lists in an automated and robust fashion. (Or, rather, we need to check to see if it needs rebuilding.)

While doing all this it would be prudent to log changes of config with detail sufficient that we can reverse any problems. If someone broadcasts a bogus net we don't want it on our server forever.

5 NNStat

5.1 Intro

These comments apply to version 3.2 of NNStat, which was the current version at the start of 1993. A beta release of version 3.3 has been recently released.

NNStat is distributed as source code for Sun and DECstation platforms. It relies on a permissive mode network interface, using the NIT device on SunOS and the packetfilter option on ULTRIX. Two daemons do most of the work. **Statspy**, the monitor process, scans packets available on the network interface and matches them against a pattern config to control various counting operations. **Collect**, the logger process, periodically interrogates the **statspy** monitor and logs the information in files. A third tool, **rspy**, allows interactive query and control of the **statspy** monitor. (**Rspy** is only used for diagnostic purposes in our system.)

Counting operations are object-based, and the **collect** logger can be told to look for all objects or given subsets of objects. Checkpoint and logging periods are constant for a given **collect** process; using several **collect** processes, one can log various objects at different temporal resolutions.

Three periods – the polling, checkpoint and clear intervals – can be set independantly for each **collect** process. At each polling period, **collect** gathers totals for its chosen objects from **statspy**, and logs them to files. Subsequent polls overwrite the previously logged record, until the checkpoint period expires, at which time the next record to be logged is appended to the file. Totals between checkpoints (and polls) are cummulative until the clear period expires, at which point all counters are reset.

5.2 Constraints of NNStat

A number of constraints are imposed by the current implementation of NNStat.

5.2.1 Single **statspy** per Server

One can only have one monitor process – and one active config – per server. Because of this we can't test configs on the production server without interrupting stats collection. We either suffer downtime or use a developmental server.

5.2.2 Polling Rewrite Bug

The method NNStat uses for the **collect** processes to log info to files is buggy: logs sometimes acquire inter-record garbage. Because of this the log files must be parsed with some non-intuitive tests during postprocessing.

5.2.3 NNStat Config Limits

There are various fixed limits in the NNStat code, some of which aren't documented. These limits can be changed by recompiling the source, but any given config must work within the limits of the currently installed executables. The limits which we know of are:

- maximum number of objects for collector
- maximum number of cases in a "select" statement
- maximum number of parameters for object class definitions

These limits have all been raised in our installation. Some config overflow errors are silently ignored, some trigger core dumps – either way, we must check that they aren't exceeded when building the config.

5.2.4 Subnet Bug

There is a bug in version 3.2 preventing access to subnet information on little-endian architectures (*eg*, DECstation). A patch for this was found by Gavin Stone-Tolcher (ccgavin@cc.uq.oz.au).

5.2.5 **Select** Bug?

There may be a bug in the number of cases recognised by the "select" statement – either that or a confusion of datatypes has led to only half the specified maximum being useable. This will be confirmed when our developmental server is configured.

5.2.6 Separate Configs for **statspy** and **collect**

The monitor process and logging processes look at separate config files for essentially the same information. This marginally complicates building the configuration.

6 Implementation Architecture

6.1 Server Platform

The production server, noc, is a DECstation 5000/250 with 64M ram and 2G disk, running ULTRIX 4.3. * While noc was being coaxed to sporadic life, the project was supported on a Solbourne S4000 with 40M ram and 1G disk, running OS/MP 4.1a.3 (licensed SunOS, ˜4.1.2).

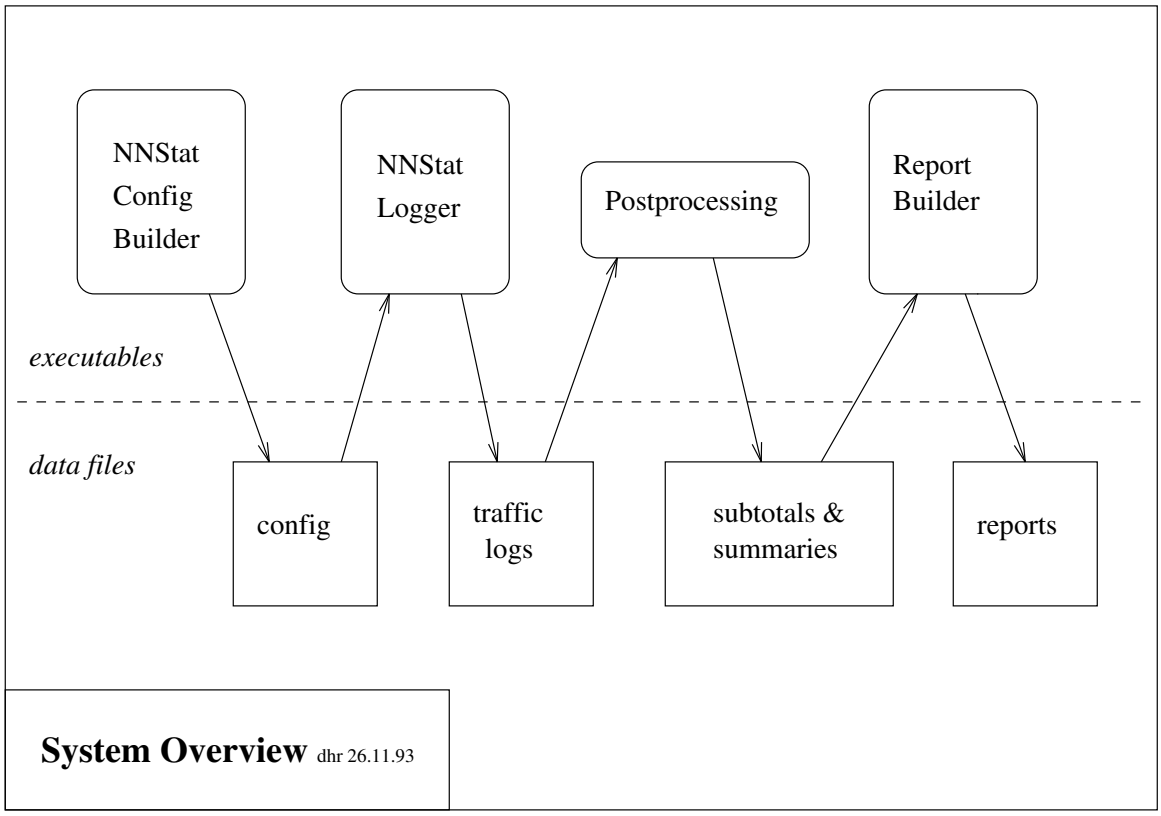
6.2 Software

The present system is cobbled together from scripts. With marginal editing of reality, it can be divided into the four functional units shown in the figure below. (Most of the NNStat traffic logger and a good portion of the postprocessing scripts were donated by Robert Elz, from his work on the AARNet stats server.)

(In the following few figures, executable modules are shown in the upper portion and data files in the lower portion of each diagram.)

---

* This is not an endorsement of DEC.

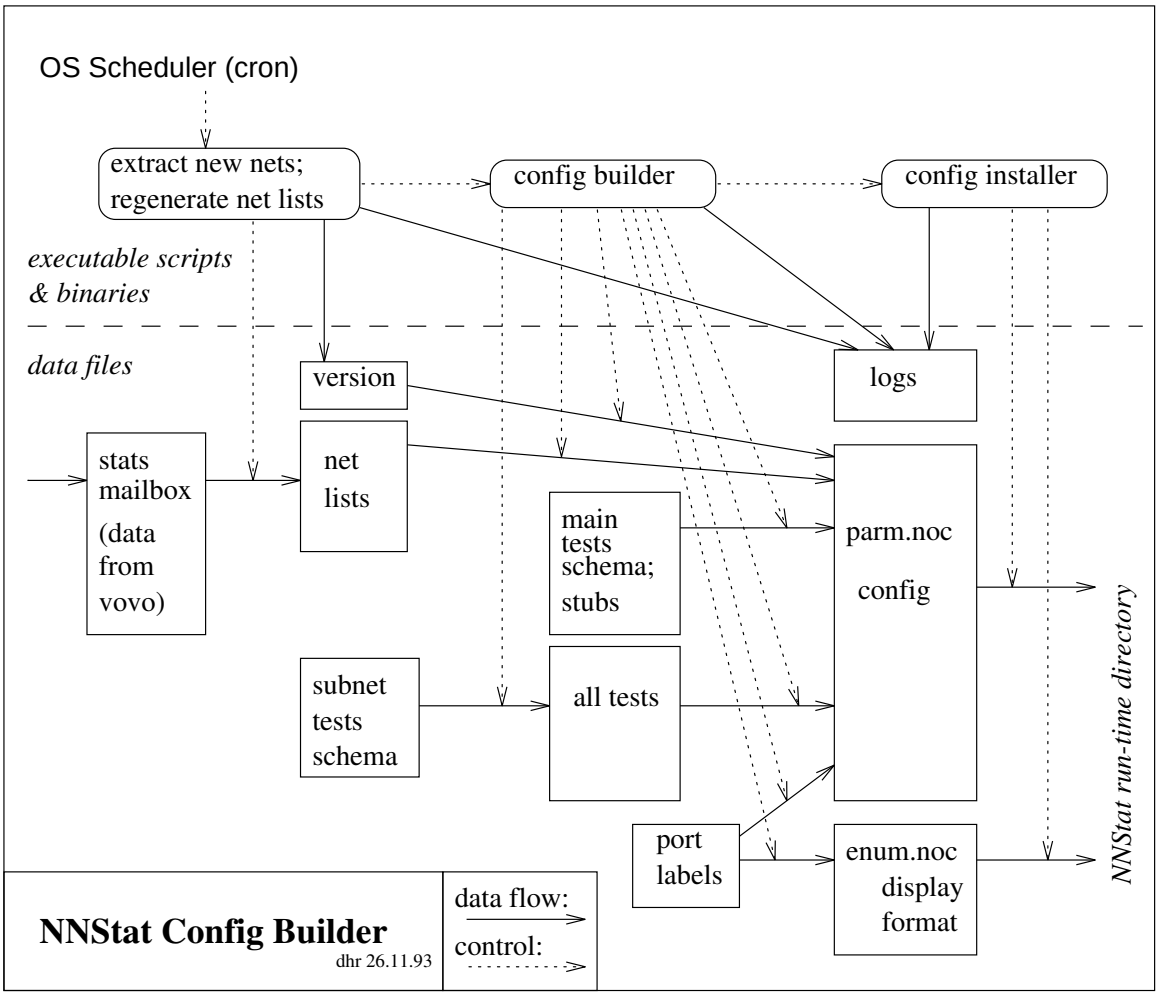


My aim is to automate the procedure as far as possible. Human intervention should only be required for qualitative evaluations, during the report generation phase. Currently, both the config creation and traffic logging run untended. Postprocessing is still initiated manually, pending rationalisation of error reporting and satisfactory error handling. Report generation facilities are meagre.

6.2.1 Config Builder

The config builder updates the NNStat config with new Victorian and other Australian network numbers. It sifts through the network lists obtained from vovo, and when new numbers are found, incorporates them into the config. It maintains version details for the config, logs which vovo file the new numbers came from, and logs when a new config is installed. The figure below gives a (rather simplified) view of the process.

There are three main parts in the code. First there's a module that manages the network numbers. This is called nightly from cron. When new numbers are found, it calls the config builder module, which reassembles the NNStat config from various schemas. Finally, if the config was rebuilt successfuly successful, the config builder calls a module which installs the config.



Integrated within this code are checks to ensure that a new config doesn't exceed NNStat's limits (insofar as these are known). If these are exceeded, the new config will not be installed, future config updates will be suspended, and mail sent to the administrator. Before this happens, when the config expands beyond certain thresholds, warning mail is sent to the administrator. This will allow a new set of executables to be compiled and installed before config updates are interrupted.

The main schema for the config implements the algorithm discussed earlier (see section 4.1). The blocks of tests referred to record subnet and type-of-service, and, for the most part, have a common structure. Accordingly, we generate all eight blocks by expansion of a single schema of subnet tests – this simplifies modifications if we need to change the level of detail of information logged.

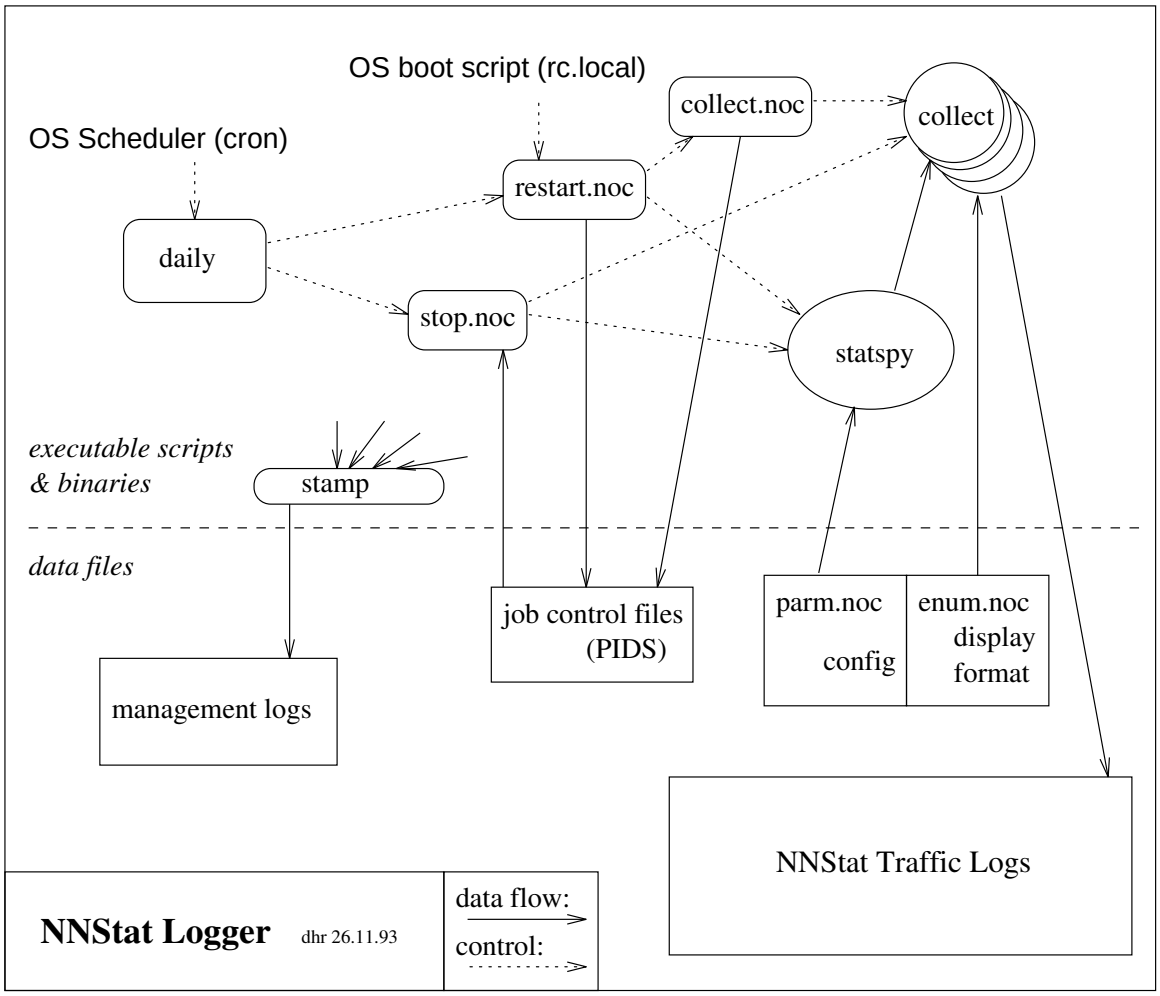
The block which records traffic into the University from Australian networks has the following form. First, the total for all incoming ethernet traffic is logged. Traffic destined for hosts on the UoM ether spine (excluding the fddi routers) is recorded, and then we deal with traffic heading for the fddi ring. For traffic bound for the FDDI ring, exceptions are handled – munnari. We keep track of munnari's traffic because it supplies national services. All IP traffic is logged, and then, separately, TCP traffic and UDP traffic. Source and destination ports are logged separately for TCP and UDP. Having dealt with munnari, we handle the rest of the University similarly, logging first IP traffic totals, then subtotals for TCP matched on source port, TCP matched on destination port, UDP matched on source port and UDP matched on destination port. The destination subnet is recorded, to give some idea of which parts of the University are in communication.

```
# traffic into campus from ausnets
record destination-subnet
select Ether.destination {
 case (  HOSTS-ON-UoM-ETHER-SPINE )
    record Ether.destination
 case ( "rb1.rtr", "rb2.rtr" ): {
    # to campus on FDDI ring:
  if IP.destination is munnari {
   record IP.destination
   if TCP.destinationport is in TCP-PORTS-LIST
     record TCP.destinationport
   if TCP.sourceport is in TCP-PORTS-LIST
     record TCP.sourceport
   if UDP.destinationport is in UDP-PORTS-LIST
     record UDP.destinationport
   if UDP.sourceport is in UDP-PORTS-LIST
     record UDP.sourceport
  }

  # All subnets (including munnari traffic):
  record IP.destination-subnet
  if TCP.destinationport is TCP-PORTS-LIST
   record TCP.destinationport, IP.destination-subnet
  if TCP.sourceport is TCP-PORTS-LIST
   record TCP.sourceport, IP.destination-subnet
  if UDP.destinationport is UDP-PORTS-LIST
   record UDP.destinationport, IP.destination-subnet
  if UDP.sourceport is UDP-PORTS-LIST
   record UDP.sourceport, IP.destination-subnet
 }
 default : {   # shouldn't be anything in here.
  record Ether.destination, Ether.source
  record IP.destination, IP.source
 }
}                    # end select Ether.dst
```

6.2.2 NNStat Traffic Logger

The traffic logger is shown (almost verbatim) in the figure below. The system is started at boot time from rc.local, and restarted at midnight from cron. There is a script to control starting the NNStat processes (restart.noc) and another which stops them (stop.noc).



The *collect* processes log information in files named after the object being logged, one file per object, and given a suffix indicating the time at which the **collect** process was invoked. The timestamp causes the **collect** processes to log to different files (unless processes are started within the same minute).

We choose to stop and restart the **statspy** and **collect** processes daily. This means that we get a new set of log files each day, which simplifies expunging corrupted logs from the data. It also makes it easy to install a new config when one is required. (We could manually update the runtime config on **statspy** via **rspy**, but as we have to generate the new config anyway, and we have an excuse for stopping the logging, there would be no point in such sophistication.)

Process IDs are recorded and timestamped whenever the processes are stopped or restarted, and a comment can be automatically inserted in the log when the scripts are run manually. When **statspy** is started it gives a commentary on what it thinks of your config. This diagnostic message is superficially parsed by the start-up script (restart.noc) to detect key words ("error", "warning"), and when problems are detected mail is sent to the administrator.

One action not represented on the diagram is the weekly transfer of the traffic logs. Once a week, between stopping and restarting the NNStat processes, the **daily** script renames the current logging directory with a timestamp and recreates the logging directory. (Our reports are based on weekly aggregates).

6.2.3 Postprocessing

6.2.3.1 Input
6.2.3.2 Output
6.2.3.3 Reconcilliation
6.2.3.4 Geographic totals pipeline

6.2.3.5 Type-of-Service totals pipeline

Under postprocessing we subsume the evils that happen after the week's data has been logged. The processing modules of this section form a number of data pipelines applicable to different groups of objects.

6.2.3.1 Input

The input, the raw traffic log files from the NNStat logger, are byte and packet totals sampled cummulatively over each day for a series of "data objects". We choose to generate weekly totals. The records for a given object will be distributed over a number of files, a new file for each time the **collect** processes are restarted.

The objects we log can be divided into two groups. Most are sampled at 30 minute intervals and checkpointed only at the 24h mark. A smaller group are sampled at 5 minute intervals and checkpointed every 15 minutes. The latter are the IP logs used for generating total traffic figures for each target geographical destination, and containt subtotals for each subnet.

Below is a sample NNStat record for information coming into campus from Australian networks. (Local subnet and subtotals have been overwritten for mystique). The first figure after the subnet key is the number of packets, and the second figure the number of bytes. "Total Count" sums the packet information.

```
OBJECT: F.IN.dstsubn-all.from-ausnets.ip  Class= freq-all-bytes [Created: 00:00:34 11-21-93]
  ReadTime: 00:00:00 11-22-93, ClearTime: 00:00:40 11-21-93 (@-86360sec)
  Total Count= 649370 (+0 orphans)
  Total Bytes= 48370339B    #bins = 43
[128.250.aaa.0]= 999999 &99999999B   (75.6%) @-0sec
[128.250.bbb.0]= 88888 &8888888B    ( 6.9%) @-17sec
[128.250.ccc.0]= 77777 &7777777B    ( 4.9%) @-0sec
[128.250.ddd.0]= 66666 &6666666B    ( 2.4%) @-0sec
[128.250.eee.0]= 55555 &5555555B    ( 2.2%) @-11sec
[128.250.fff.0]= 44444 &444444B     ( 1.8%) @-7sec
 ...
```

The former, less frequently sampled group contain objects logging type-of-service details for TCP and UDP ports for each subnet. Below is a sample recording TCP source-port details for traffic from Australian networks into campus. (Local subnet and subtotals have been overwritten from boredom).

```
OBJECT: S.IN.dstsubn.from-aus.tcp.srcports  Class= matrix-all-bytes  [Created: 00:00:34 11-21-93]
  ReadTime: 00:00:00 11-22-93, ClearTime: 00:00:41 11-21-93 (@-86359sec)
  Total Count= 412647 (+0 orphans)
  Total Bytes= 25095824B    #bins = 45
[119 "NNTP" : 128.250.aaa.0]= 999999 &99999999B  (90.8%) @-0sec
[513 "rlogin|rwho" : 128.250.bbb.0]= 88888 &888888B ( 2.7%) @-21797sec
[23 "Telnet" : 128.250.ccc.0]= 7777 &7777777B   ( 1.6%) @-21090sec
[25 "SMTP" : 128.250.ddd.0]= 6666 &666666B       ( 1.2%) @-21sec
[20 "FTP data" : 128.250.eee.0]= 5555 &5555555B    ( 1.0%) @-7766sec
[20 "FTP data" : 128.250.fff.0]= 4444 &4444444B ( 1.0%) @-12179sec
[79 "Finger" : 128.250.ggg.0]= 3333 &333333B   ( 0.4%) @-6410sec
 ...
```

6.2.3.2 Output

For each object we generate a single NNStat-style record sumarising the weeks traffic.

6.2.3.3 Reconcilliation

As shown above, each NNStat record contains a header of summary info, and a table of values. When ever our postprocessing modules read or write an NNStat record, they verify that the byte and packet totals in the header are still within a small percentage of the actual sums of the table. (Anomalies are reported and

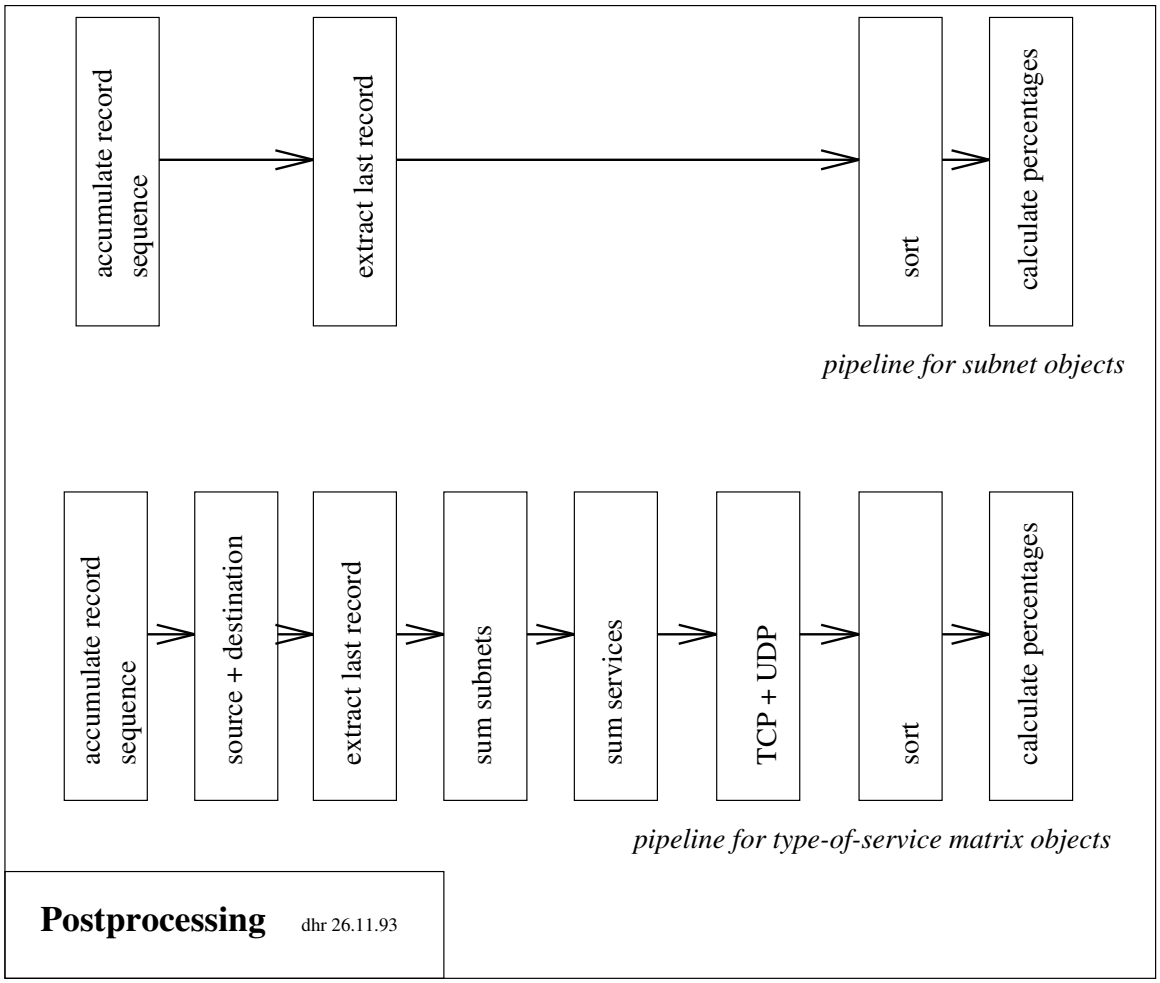
can be detected by simple string searches of the postprocessing log.) This simple reconcilliation procedure has detected a number of instances where corrupted data slipped past the initial log parsing. However, it is probably not foolproof. It would be safer if further reconcilliation phases were introduced within the pipeline. A good target would be ensuring that the sum of IP traffic is not less than the sum of all TCP and UDP services.

6.2.3.4 Geographic totals pipeline

Processing for simple sums of IP to and from of the geographic target areas are conceptually straightforward.

All the records relevant to a given object are collated and the effects of any restarts or clears between records are removed. The last record is then the accumulation of the weeks data, so this record is extracted. The results are sorted, with separate tables being produced for packet and byte measures.

This forms our basic pipeline.



6.2.3.5 Type-of-Service totals pipeline

A number of extra modules must be inserted into the above unit to handle type-of-service data. Information collected for source and destination ports must be combined. Most protocols can be adequately handled in one of two ways, depending on whether they establish a connection with a well-known port number at one end (sum source and destination values) or both ends (take the maximum of source and destination values)*. At this point we have possibly several entries for a single subnet, so we must sum matching subnet/port combinations. Various services use multiple ports, so we have another module to sum these. Finally, TCP and UDP information is combined, and then the sorting modules are invoked.

---

* thanks for this rule of thumb due to Robert Elz

7 Performance Considerations

On our platforms the NNStat processes appear to be CPU-limited. The figures below are from the production server, a DECstation 5000/250. They describe performance under the current config, which sets the resolution of local peer to be at the level of subnet.

The current config logs something over 30M of files per week. The files produced by postprocessing are less than 2M. This is composed of 541 objects checkpointed daily, and 144 objects checkpointed at 15 minute intervals. The latter form more than 90% of the bulk. Any time-based accounting could substantially increase disk usage.

For the **collect** processes, size in virtual memory and the resident set size appears to be static and easily manageable. CPU time is dependant on the number of objects and amount of information per object, but the total CPU time expended over a 24h period is quite modest.

```
USER       PID %CPU %MEM   SZ  RSS TT STAT    TIME COMMAND
ray      27256  0.0  0.4 1188  200 ?  S      0:00 collect -m 644 ...
ray      27248  0.0  0.4 1188  220 ?  S      0:28 collect -m 644 ...
ray      27240  0.0  0.4 1188  220 ?  S      0:28 collect -m 644 ...
ray      27231  0.0  0.4 1188  216 ?  I      1:19 collect -m 644 ...
```

Of the above processes, the first is collecting information for one object. The middle two are each collecting information on 270 objects. The lower one is collecting information on 144 objects polled at 5 minute intervals; the upper three are polling **statspy** at 30 minute intervals.

The **statspy** process consumes appreciable CPU time, and expands gradually throughout the logging period. Memory will probably not become a problem, given that we're restarting the process every 24h – the process size starts at less than 2M (the snapshot below is 60" after invocation):

```
USER       PID %CPU %MEM   SZ  RSS TT STAT    TIME COMMAND
ray       7495  0.4  2.6 1952 1448 ?  S      0:11 statspy parm.noc
```

and on an average day grew to something over 2M:

```
USER       PID %CPU %MEM   SZ  RSS TT STAT    TIME COMMAND
ray      27219  0.0  3.1 2252 1712 ?  S N  149:33 statspy parm.noc
```

However, it consumed some 9000 seconds of CPU time in the period. Process accounting has shown this to be the mode, but there are occaisional peaks of 25,000 seconds, which is a substantial portion of the 86,400 seconds of one day. (It would be interesting to see what memory consumption had got to on those days, but we've only just started logging this.)

There's three things **statspy** might be spending CPU time on: incrementing counters; working out which counter to increment; or downloading figures to the **collect** process.

We haven't observed a marked increase in **statspy**'s CPU consumption during the periodic polls by the **collect** processes. This implies that CPU usage is basically spent in processing traffic, and we presume dominated by the complexity of pattern matching encoded in the config (*ie*, working out which counter to increment). If this is the case then, at peak traffic loads, our current server probably won't cope with a substantially more complicated config.

We have been using the ULTRIX packet filter with the default maximum queue of 32 packets (NNStat automatically requests maximum queue length). (We haven't checked what diagnostics we'd receive from either NNStat or the kernel in the event of an overflow, but will investigate this as soon as the development server is configured.)

As mooted in the initial discussion of our task (see section 1), subnet resolution may not be sufficient for some applications. Converting the config from the subnet paradigm to a departmental paradigm – moving the peer resolution towards individual hosts – dramatically increases the complexity of the config, in the worst case by more than two orders of magnitude. Clearly we can't ask CPU time to increase by this factor. (Nor, for that matter, disk usage, but we can collate host info without logging it to file.)

It remains to be seen whether some intermediate solution is plausible. Answering this reliably requires determining the CPU cost factors of various formats of config. Note that the actual CPU usage under any config depends on the profile and quantity of traffic. If reliable estimates of CPU costs are determined,

then the worst case traffic profiles and maximum traffic loads that a stats server will cope with can be predicted. (Which could be put another way – that any stats server is only reliable up to a given load of traffic, and it would probably be useful to know what that load is :)

8 Ancillary Info

Here are things we should be doing but aren't, or are still developing.

8.1 Time

A network stats server must have a reliable time source, particularly if the information collected is used to generate bills. Installing the NTP suite is a suitable solution.

8.2 System Downtime

What we miss is as important as what we get. Every report must note server downtime, and, where possible, estimate the effect of this on recorded results. Even merely registering downtime is a messy problem. We identify three components: downtime due to

- platform failure (OS or hardware);
- stats system failure (s/w error or resource privation)
- interruptions in network connectivity.

Each of these requires its own method of monitoring and logging.

8.3 Anomaly Tagging

There are a range of situations that are useful to keep in mind when viewing stats – *eg*, interruptions to logging, network storms, partial network failures, connection of new services, and changes in filtering policy. This information tends not to be collated anywhere. It is useful not only to log these events consistently, but to have some way of tagging potentially anomalous data with cross-references to the related events.

8.4 Config Version Tagging

Unfortunately, different postprocessing functionality is sometimes required in different config versions. Most simply, when new objects are logged, they must be processed. Changes in the config must be logged. For any project that will last longer than a few months it would be worth considering using the config version to tag the data, and enabling automatic selection of an appropriate postprocessing regime.

9 Conclusions

Nothing is as simple as it seems. Even given perfect network stats software, some ancillary information is required (*eg*, downtime). As far as possible, inclusion of ancillary information should be automated, as it is inordinately fiddly and time-consuming to collate manually.

It has proved to be practical to log type-of-service information, with the local side resolved to the subnet, for a class B network with class C netmask.

Subnet-level resolution is probably insufficient for billing purposes, particularly if billing at a departmental level rather than faculty level.

It is not known whether increasing resolution of local peer to the level of local hosts is practical on the current server platform. This can be investigated when the developmental server is configured.

The primary restrictions on expanding the stats system are expected to be CPU time and disk space.